计算机科学
COMPUTER SCIENCE

面向龙芯处理器的一种CompCert可信编译器重定向实现

胡少儒, 王隽伟, 王生原

引用本文

胡少儒, 王隽伟, 王生原. 面向龙芯处理器的一种CompCert可信编译器重定向实现[J]. 计算机科学, 2024, 51(11A): 240200115-9.

HU Shaoru, WANG Juanwei, WANG Shengyuan. Implementation of Retargeting CompCert Trusted Compiler for Loongson Processors [J]. Computer Science, 2024, 51(11A): 240200115-9.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于MLIR的FP8量化模拟与推理内存优化

FP8 Quantization and Inference Memory Optimization Based on MLIR

计算机科学, 2024, 51(9): 112-120. https://doi.org/10.11896/jsjkx.230900143

基于特征重要性的深度学习自动调度优化研究

Study on Deep Learning Automatic Scheduling Optimization Based on Feature Importance

计算机科学, 2024, 51(7): 22-28. https://doi.org/10.11896/jsjkx.230500220

面向TPU粗粒度指令的自动张量化方法

Automatic Tensorization for TPU Coarse-grained Instructions

计算机科学, 2024, 51(6): 52-60. https://doi.org/10.11896/jsjkx.230800049

一种基于指令MKS的自动向量化代价模型

Auto-vectorization Cost Model Based on Instruction MKS

计算机科学, 2024, 51(4): 78-85. https://doi.org/10.11896/jsjkx.230200024

编译支持的程序栈空间布局运行时随机化方法

Compiler-supported Program Stack Space Layout Runtime Randomization Method

计算机科学, 2023, 50(8): 314-320. https://doi.org/10.11896/jsjkx.220800098

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240200115

面向龙芯处理器的一种 CompCert 可信编译器重定向实现

胡少儒 王隽伟 王生原
清华大学计算机系 北京 100084
(cn.hushaoru@gmail.com)

摘 要 CompCert 是著名的 C 语言可信编译器,它借助于交互式定理证明工具 Coq 实现,能够确保生成的目标汇编代码保持源代码的语义,具有极高的可信度,近年来被广泛应用于学术界和工业界的许多安全攸关任务的研发工作中。CompCert 编译器的当前版本支持多种目标机结构,然而目前尚缺乏针对国内自主研发处理器的版本,如龙芯(Loongson)处理器体系结构(LoongArch)。将 CompCert 重定向到龙芯等国产处理器,对我国安全攸关软件领域的发展大有裨益。本文对 CompCert 编译器的设计理念、框架结构和龙芯架构的特点进行分析,改造 CompCert 编译器的后端,使其可以生成能在龙芯处理器上运行的汇编代码,并细致阐述不同模块的工作内容。重定向到龙芯处理器的 CompCert 编译器具有接近 GCC-O1 的性能,可满足许多场景的使用。

关键词:CompCert;编译器;编译器重定向;龙芯架构;形式化验证的编译器;Coq

中图分类号 TP314

Implementation of Retargeting CompCert Trusted Compiler for Loongson Processors

HU Shaoru,WANG Juanwei and WANG Shengyuan
Department of Computer Science and Technology,Tsinghua University,Beijing 100084,China

Abstract CompCert is a well-known trustworthy C-language compiler,which is highly credible and has been widely used in many research and development work in academia and industry in recent years. CompCert proves the property that the target assembly code it produces can keep the semantics of the source code,with Coq,an interactive proof assistant. The CompCert compiler now supports multiple target machine architectures,but there is currently a lack of versions specifically designed for domestically developed processors,such as the Loongson processor architecture(LoongArch). Retargetting CompCert to domestic processors such as Loongson is of great benefit to the development of safety-critical software in China. This paper analyzes the design and the structure of the CompCert compiler backend and the characteristics of Loongarch,revises the backend of the CompCert compiler to make it available to generate the assembly code suitable to run on the Loongson processor,and presents the work of the different modules. The revised CompCert compiler,which retargets to Loongson processors,has performance competitive with GCC at optimization level 1,and can meet the needs of various scenarios.

Keywords CompCert,Compiler,Compiler retargeting,Loongarch,Formally verified compilers,Coq

1 引言

计算机技术被日益广泛地应用于航空航天、高速铁路、核电能源和医疗卫生等领域的安全攸关系统(Safety-critical System),此类系统一旦失效,将给人类生命财产、社会生产、生活环境带来巨大的危险。现代计算机技术的发展中,硬件方面的安全性保障技术相比软件来说要更加成熟,软件方面的安全一直是计算机系统安全性中十分薄弱的环节。

软件安全性最终体现在可执行目标程序/代码的安全可信上。从软件生产环节到软件代码的执行,各阶段的代码生成器或编译器是影响可执行代码安全可信的直接因素。首先,对于安全攸关系统而言,必须考虑编译器引入的错误,否则花大力气在源程序级的验证工作上可能在目标程序级失效。实际上,如航空领域的 RTCA DO-178B/C 标准,编译器属于需要鉴定的工具类软件,需要按照机载软件的要求对待。

可信编译器(Trustworthy/Credible/Dependable/Reliable Compiler)主要包含两层含义:编译器自身的安全可信,以及尽可能保障被编译对象的安全可信。本文主要关注前者。

所谓编译器自身的安全可信,就是要保证编译器做正确的事情,保证它能将符合源语言规范的程序正确翻译到目标语言程序。这里,“正确翻译”主要指功能方面,即保证源语言的行为特征能够在目标语言中被正确地体现,目标程序的行为能够准确反映其对应源程序的行为。不能实现“正确翻译”的编译器,则一定存在某种程度的“误编译”错误。

为了保证编译器自身的安全和可信,杜绝“误编译”,传统的方法是采用测试和严格的过程管理。例如,GCC 的 torture 测试集包含几千个 C 源程序用例,商用的 Plum Hall Standard Validation Suite for C 有几万个用例。还有一些 Bug-

基金项目:国家重点研发计划(2022YFB3305204)
This work was supported by the National Key R&D Program of China(2022YFB3305204).
通信作者:王生原(wwssyy@tsinghua.edu.cn)

hunting 工具,例如 Csmith[1],它可以产生更多或更独特的源程序用例,从而覆盖更广,发现更多的编译器缺陷。实际上,仅编译器自动测试工具 Csmith(截至 2011 年 2 月),以及 EMI/SPE[2](截至 2017 年 10 月份),就发现了 GCC 和 LLVM 的共计近 2 000 个“误编译”错误。尽管如此,采用测试的手段是不可能完善的,即便是通过测试发现了错误并且做了修改,也无法保证编译器自身的正确性。

为进一步增加编译器的安全和可信程度,需要对所关注的重要性质进行正确性验证,相当于全覆盖测试,这是解决问题的根本途径。最严格的验证手段莫过于采用形式化方法。工业界也早已意识到形式化软件开发的潜力,在一些安全攸关领域的安全级软件开发标准中也逐步新增了形式化方法相关的目标或者相应的补充说明。CC(Common Criteria)安全评估标准中将可信性分为 7 个级别(EAL1 到 EAL7),可信性级别越高,其采用形式化规范和验证的程度就越高。航空无线电委员会(RTCA)近期也已推出民航电子系统的国际标准 DO-178C,其相比 DO-178B 增加了对于形式化规范和验证的要求,如 DO-333 等补充说明。

人们在几十年前就开展了编译器形式化验证的工作[3-5],包含从简单语言的单遍编译器到较成熟的代码优化遍历等形形色色的工作。近年来,技术不断进步,已经可以验证较为复杂的编译器,其中的杰出代表为 Leroy 等的 C 可信编译器 CompCert[6-7]。

编译器形式化验证的两种主流技术为:1)对编译过程本身进行形式化验证,多采用构造即证明的方法(Correct-by-Construction),得到经过验证的编译器(Certified/Verified Compiler);2)翻译确认(Translation Validation)[8],构造可信的翻译前后确认程序(Validator),确认程序的构造通常基于静态分析、模型检查、约束求解等自动分析或证明技术,必要时可进一步对确认程序进行验证。

CompCert 是首个可商用的经过形式化验证的 C 语言可信编译器(2015 年起),于 2008 年 3 月发布首个版本(CompCert C Compiler 1.2),获得了 2016 年度的十年最有影响 POPL 论文奖、2021 年度 ACM 系统软件奖(ACM Software System Award,公认的计算机科学研究界最高软件奖)以及 2022 年度 SIGPLAN 编程语言软件奖(ACM SIGPLAN Programming Languages Software Award)。

如图 1(源自文献[9])所示,CompCert 编译器的源语言为 CompCert C,是 C 语言的大子集,接近于支持/兼容 ISO C 标准 C99/C2011。经过前端解析(含词法和语法分析)、符合性检查(各种静态检查)以及规范化(剔除表达式计算中的副作用,明确求值顺序等 ISO C 标准中一些模糊的内容),编译器将 CompCert C 变换为一种简化的中间表示 Clight,后者已消除表达式的副作用,满足语义确定性。从 Clight 开始,编译器经历另外 8 层中间表示以及 10 多遍变换,最终生成目标汇编代码。当前版本支持 PowerPC,ARM,RISC-V 和 x86 等目标处理器(含 32 位和 64 位处理器)。CompCert 编译器的优化性能接近于 GCC 4(-O1),能够满足大多数嵌入式系统的应用需求。

CompCert 最突出的特点就是其大部分实现是在证明辅助器 Coq 环境中完成的,并且除词法分析和某些预处理过程之外,各个翻译阶段均在 Coq 中实现了正确性证明。从 Clight 开始到目标汇编代码的生成,是 CompCert 编译器实现及其形式化验证最核心的部分,证明了所生成汇编代码保持了 Clight 源代码的语义。在 CompCert 中,所证明的语义保持性质可描述为一种正向的行为模拟等价关系。从 Clight 至 Asm AST 将近 20 遍翻译中,除寄存器分配外的其余各遍均是直接证明翻译过程本身满足这种语义保持性质。这些中间语言的(动态)语义均定义为操作语义(Operational Semantics),且基于统一的内存模型(Memory Model)[10]。

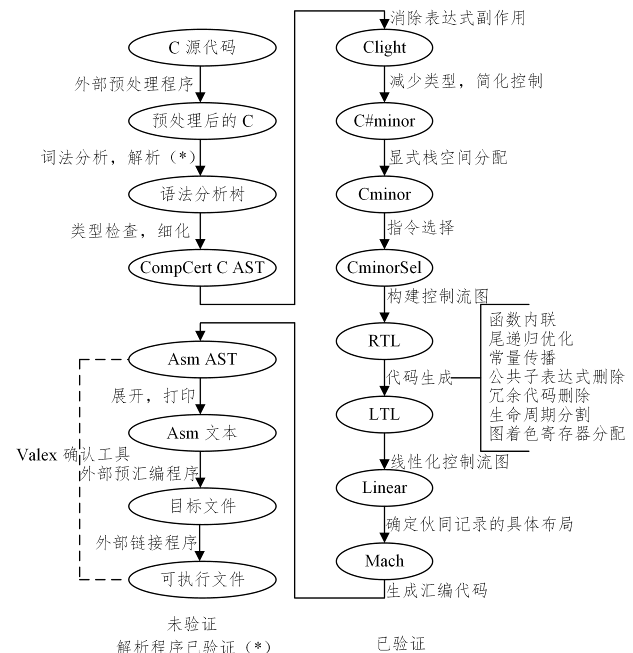


图 1 可信编译器 CompCert 架构

Fig. 1 Architecture of trustworthy compiler CompCert

将 CompCert 重定向到国产处理器,对我国安全攸关软件领域的发展大有裨益。本文尝试将 CompCert 重定向到国产龙芯(Loongson)处理器体系结构(LoongArch)。第 2 章概要介绍 CompCert 的后端结构;第 3 章针对龙芯处理器的指令系统架构对整体后端设计方案进行简要介绍;第 4 章着重介绍相关模块的实现;第 5 章通过性能测试比较 CompCert 与 GCC 生成的代码的效率;最后对整体设计和实验结果进行总结。

2 CompCert 后端结构

参见图 1 的示意图,CompCert 编译器从 Clight 层中间表示开始逐步增加低级特性。首先是细化访存操作及部分局部量的显式栈空间分配,体现在中间表示 C♯minor 和 Cminor。CompCert 编译器中 Cminor 之后的部分称为后端(backend)。后端总计有 12 种变换,涉及 8 种中间语言,目的是将 Cminor 翻译为目标处理器的汇编语言。从 Cminor 到 CminorSel,生成了面向特定机器的算术/逻辑运算、布尔条件以及寻址方式。至中间表示 RTL,程序已转换为了基于三地址码、伪寄存器(不限量的中间变量)的控制流图结构。从 RTL 到 LTL 的变换,程序变换为以三地址码基本块为节点的控制流图,且通过图着色寄存器分配将伪寄存器访问转换为有限个数的物理寄存器访问(并未指定具体处理器)。接着,从

LTL 到 Linear 的变换，控制流图被替换为含显式的分支和标号的线性指令列表。然后，从 Linear 到 Mach 的变换，使得活动记录的访问操作更加具体化，使用了实际偏移量而非抽象的栈帧单元的位置，同时也显式区分开了 caller 和 callee 的栈帧空间。Mach 与汇编语言已经非常接近，只需要根据处理器的具体指令集、伪指令的具体表示形式等对代码序列进行转换，即可得到汇编语言的抽象语法树，供之后翻译成文本并进行汇编。

除上述编译主体的各阶段翻译过程，CompCert 编译器也包含了某些层次的中间代码优化工作。各级中间语言在编译器中的关联关系进一步参见文献[11]。

从 Clight 开始直到目标汇编代码的生成，是 CompCert 编译器实现及其形式化验证最核心的部分，证明了所生成汇编代码保持了 Clight 源代码的语义。对于 CompCert 大多数翻译遍，所证明的语义保持性质可描述为一种正向的行为模拟等价关系[6-7,12]：

$\forall B \notin \text{Wrong}, S \Downarrow B \Rightarrow C \Downarrow B$

这里，S 和 C 分别代表当前翻译过程的源和目标程序，表示满足安全行为的语义。之所以上式所描述的正向行为模拟等价关系可以刻画翻译过程的语义保持行为（通常认为是一种反向的行为模拟等价关系），是建立在这一前提之上：Clight 以及其后续中间语言的语义均满足确定性。

唯一例外是从 RTL 翻译至 LTL 的寄存器分配过程，它采用一种改进的图着色寄存器分配算法[13]，其正确性验证在目前的开源版 CompCert 中采用翻译确认[8]且同时证明该确认程序正确性的方法实现[14]。虽然该方法不能保证确认程序不会出现误报（通常，采用翻译确认方法均会有一定的误报率），但文献[14]提到在作者的实际经验里从未遇到过发生误报的情况。

设 S 和 C 分别代表当前翻译过程的源和目标程序，Comp 为翻译函数，则带确认程序 Validate 的翻译函数 Comp′可定义为

```
Comp′(S)=
  match Comp(S) with
    |Error →Error
    |OK(C) →if Validate(S,C) then OK(C) else Error
  end.
```

这样，确认程序 Validate 的正确性可描述为

$\forall\ B, C, Validate(S,C)=\text{true} \Rightarrow S \approx C$

其中，$S \approx C$ iff $\forall B, B \notin \text{Wrong} \wedge S \Downarrow B \Rightarrow C \Downarrow B$。

CompCert 团队的另一工作[15]是对这种改进的图着色寄存器分配算法进行直接验证，该方法具有完备性，可证明算法实现的完全正确性。即证明

$\forall\ B, C, Comp(S)=OK(C) \Rightarrow S \approx C$

图 1 中 Asm AST 之后的步骤均为进行形式化验证，但 CompCert 提供了一个从 Asm AST 至可执行文件的翻译确认工具 Valex，可用于与这些步骤的验证。然而，CompCert 并未实现工具 Valex 本身的正确性证明。

3 重定向设计方案

3.1 龙芯处理器与龙芯架构

龙芯处理器起源于中国科学院计算技术研究所。2002 年，“龙芯一号”处理器诞生，当时采用 MIPS 指令集。在二十年的技术积累与生态建设的基础上，龙芯中科于 2020 年推出了龙芯架构（LoongArch）。龙芯架构的指令包含基础部分和扩展部分，总共有接近 2000 条指令[16]。2021 年 4 月 15 日，龙芯架构正式对外公布。2021 年 7 月 23 日，龙芯中科正式发布“龙芯 3A5000”，这是基于龙芯架构研发的第一代国产处理器。龙芯架构采纳了许多先进的技术，在软硬件的性能提升方面都卓有成效；同时，龙芯架构融合了其他主流指令系统的一些功能特性，可以做到高效率的二进制翻译[17]。目前，龙芯系列设备已经广泛用于许多专业领域。

龙芯架构是 RISC（Reduced Instruction Set Computing）风格的指令系统架构。龙芯架构有固定的指令长度和规整的编码格式，大部分指令包含两个源操作数和一个目的操作数，并且除了 load/store 指令可以直接访问内存外，其他指令只能操作立即数或寄存器。龙芯架构基础部分的非特权指令集按照软件运行时上下文内容的差异可划分为基础整数指令和基础浮点数指令两个部分。基础整数指令涉及的寄存器包括通用寄存器（General Register）和程序计数器（Program Counter）。基础浮点数指令涉及的寄存器包括浮点寄存器（Floating-point Register）、条件标志寄存器（Condition Flag Register）和浮点控制寄存器（Floating-point Control and Status Register）[18]。

3.2 重定向需要修改的模块

CompCert 编译器各阶段所生成的中间代码从 CminorSel 层（见图 1）便涉及目标处理其特定的运算、布尔条件以及寻址方式，以及在后续的各个层次也用到了目标处理器特定的其他信息。尽管如此，CompCert 的后端组织将面向目标处理器结构的相关模块独立出来，有益于编译器的重定向开发[11]。

CompCert 编译器已经在其他架构上有了完整的实现，可以基于已经设计好的代码架构对 CompCert 编译器进行重定向。CompCert 的前端代码与后端架构无关，因此任何后端都可以直接使用；CompCert 后端代码部分也同样包含一些通用代码（位于 backend/等文件夹下）。重定向到龙芯架构后端时，只需要新建 loongarch/文件夹，提供龙芯处理器架构相关的模块，并在一些配置相关的文件中增加龙芯的配置项。

为使整个开发过程变得顺利，本文工作很大程度上借鉴了 RISC-V 后端的设计。一方面，CompCert 本身架构要求后端提供对应的模块接口，基于已有的完整代码会让开发变得比较容易；另一方面，龙芯架构和 RISC-V 架构也比较接近，二者都是 RISC 风格的指令系统架构[19]。

分析 CompCert 源代码以及参考文献[11]，若将 CompCert 后端重定向到支持龙芯架构，需要在 loongarch/文件夹下提供 Op. v，SelectOp. v，SelectLong. v，SelectOpproof. v，SelectLongproof. v，ValueAOp. v，ConstpropOp. v，ConstpropOpproof. v，CombineOp. v，CombineOpproof. v，NeedOp. v，Machregs. v，Conventions1. v，Stacklayout. v，Asm. v，Asmgen. v，Asmgenproof1. v，Asmgenproof. v，Archi. v，Asmexpand. ml，以及 TargetPrinter. ml 等代码模块文件，所需其余模块则可共享位于 backend/文件夹下的后端通用代码。

需要修改的代码模块大致可以分为 5 类（省略证明模块、

调试相关的模块和配置相关的模块)：

1)机器描述相关的模块，包括 Archi. v 和 Machregs. v。二者定义了目标处理器架构相关的参数，如 Archi. v 定义了大小端、位数等信息，Machregs. v 定义了可分配的整数寄存器和浮点寄存器。

2)调用约定相关的模块，包括 Conventions1. v 和 Stacklayout. v。Conventions1. v 定义了寄存器约定和函数调用约定，Stacklayout. v 定义活动记录的结构。

3)指令选择相关的模块，包括 Op. v，SelectOp. v，SelectLong. v。它们工作在 Cminor 到 CminorSel 变换阶段，定义面向目标处理器的相关操作符，并提供相关函数辅助 Cminor 语言向这些操作符的翻译。

4)中间代码优化相关的模块，包括 ValueAOp. v，ConstpropOp. v，CombineOp. v，NeedOp. v。它们会参与 RTL 到 LTL 变换阶段的优化工作。优化相关的算法大多由通用模块完成，这里主要是描述操作符的性质。

5)生成目标汇编相关的模块，包括 Asm. v，Asmgen. v，Asmexpand. ml，TargetPrinter. ml。它们工作在后端的最后一个阶段，即 Mach 到 Asm 变换阶段。Asm. v 对具体寄存器、汇编指令进行定义，Asmexpand. ml 对部分伪指令进行展开，Asmgen. v 将 Mach 语言翻译为汇编，TargetPrinter. ml 进行最后的格式化输出。

Power PC 结构	ARM ARM 64位 X86（32位） X86（64位） ……	RISC-V 结构
Op SelectOp SelectLong Machregs ConstpropOp CombineOp ValueAOp NeedOp Archi Conventions1 Stacklayout Asmgen Asm ...		Op SelectOp SelectLong Machregs ConstpropOp CombineOp ValueAOp NeedOp Archi Conventions1 Stacklayout Asmgen Asm ...

图 2 CompCert 中含处理器结构信息的主要模块

Fig. 2 Main modules with processor architecture information in CompCert

目前，龙芯架构的相关文档还不够详尽，没有列出部分伪指令，也没有提及部分 C 语言语法的翻译方式。本文的具体开发工作，参考了 GCC 编译器的龙芯架构实现，通过观察 GCC 对特定 C 源代码的编译表现来确定相关细节。同时，GCC 编译器也在之后用于性能对比。

3.3 加法指令示例

下面以 32 位寄存器加法指令为例，说明翻译一条指令具体需要进行的工作。

1)关于指令选择，backend/Selection. v 用到 Op. v，SelectOp. v，SelectLong. v 等模块提供的一系列函数，帮助翻译 Cminor 语言。比如 Selection. v 会借助 add 函数翻译 Cminor. Oadd。首先，Op. v 需要定义 Oadd 等运算符，在 eval_operation 中定义其操作语义，在 type_of_operation 中定义其输入和输出类型。之后，SelectOp. v 中提供 add 函数，在对进行操作的表达式进行适当分析后，选择合适的运算符进行输出。如图 3 所示。

```
(* Op.v *)
Inductive operation : Type :=
| Oadd.
Definition eval_operation
(F V: Type) (genv: Genv.t F V) (sp: val)
(op: operation) (vl: list val) (m: mem): option val :=
match op, vl with
| Oadd, v1 :: v2 :: nil => Some (Val.add v1 v2)
end.
Definition type_of_operation (op: operation) : list typ*typ :=
match op with
| Oadd => (Tint :: Tint :: nil, Tint)
end.
(* SelectOp.vp *)
Nondetfunction add (e1: expr) (e2: expr) :=
match e1, e2 with
| Eop (Ointconst n1) Enil, t2 => addimm n1 t2
| t1, Eop (Ointconst n2) Enil => addimm n2 t1
| Eop (Oaddimm n1) (t1:::Enil), Eop (Oaddimm n2) (t2:::Enil)
=> addimm (Int.add n1 n2) (Eop Oadd (t1:::t2:::Enil))
| Eop (Oaddimm n1) (t1:::Enil), Eop (Oaddrstack n2) Enil =>
Eop Oadd (Eop (Oaddrstack (Ptrofs.add (Ptrofs.of_int n1)
n2)) Enil ::: t1 ::: Enil)
| Eop (Oaddrstack n1) Enil, Eop (Oaddimm n2) (t2:::Enil) =>
Eop Oadd (Eop (Oaddrstack (Ptrofs.add n1 (Ptrofs.of_int
n2))) Enil ::: t2 ::: Enil)
| Eop (Oaddimm n1) (t1:::Enil), t2 =>
addimm n1 (Eop Oadd (t1:::t2:::Enil))
| t1, Eop (Oaddimm n2) (t2::Enil) =>
addimm n2 (Eop Oadd (t1:::t2:::Enil))
| _, _ => Eop Oadd (e1:::e2:::Enil)
end.
```

图 3 32 位寄存器加法指令涉及的指令选择

Fig. 3 Instruction selection for 32-bit register add instruction

2)对应于中间代码优化，ValueAOp. v，ConstpropOp. v，NeedOp. v 中会对 Oadd 的性质进行描述。如图 4 所示。

```
(* ValueAOp.v *)
Definition eval_static_operation
(op: operation) (vl: list aval): aval :=
match op, vl with
| Oadd, v1::v2::nil => add v1 v2
end.
(* NeedOp.v *)
Definition needs_of_operation
(op: operation) (nv: nval): list nval :=
match op with
| Oadd => op2 (modarith nv)
end.
(* ConstpropOp.vp *)
Nondetfunction op_strength_reduction
(op: operation) (args: list reg) (vl: list aval) :=
match op, args, vl with
| Oadd, r1::r2::nil, I n1::v2::nil => make_addimm n1 r2
| Oadd, r1::r2::nil, v1::I n2::nil => make_addimm n2 r1
end.
```

注：这里与 CombineOp. v 无关，CombineOp. v 主要是处理一些与立即数相关的指令。

图 4 32 位寄存器加法指令的中间代码优化

Fig. 4 Intermediate code optimization for 32-bit register add instruction

3)对应于目标汇编代码生成，Asm. v 根据具体的汇编指令定义 Paddw 指令，并在 exec_instr 中定义其操作语义。Asmgen. v 的 transl_op 函数处理 Op. v 中的运算符，将 Oadd 翻译为 Paddw。TargetPrinter. ml 的 print_instruction 函数将 Paddw 格式化输出。如图 5 所示。

```
(* Asm.v *)
Inductive instruction : Type :=
| Paddw (rd: ireg) (rj rk: ireg0).
Definition exec_instr (f: function) (i: instruction) (rs:
regset) (m: mem) : outcome :=
match i with
| Paddw d s1 s2 =>
Next (nextinstr (rs#d <- (Val.add rs##s1 rs##s2))) m
end.
(* Asmgen.v *)
Definition transl_op (op: operation) (args: list mreg) (res:
mreg) (k: code) :=
match op, args with
| Oadd, a1 :: a2 :: nil =>
do rd <- ireg_of res;
do rs1 <- ireg_of a1; do rs2 <- ireg_of a2;
OK (Paddw rd rs1 rs2 :: k)
end.
(* TargetPrinter.ml *)
let print_instruction oc = function
| Paddw(rd, rj, rk) ->
fprintf oc "add.w %a, %a, %a\n" ireg rd ireg0 rj ireg0 rk
```

图 5　32 位寄存器加法指令的汇编指令生成

Fig. 5　Assembly code generation for 32-bit register add instruction

4　实现

本章从机器描述、调用约定、指令选择、中间代码优化、生成目标汇编、可变参数、配置文件、测试不同功能出发，对重定向需要编写的不同模块的内容和作用进行细致分析与介绍。在涉及到较为复杂的情形时，会给出相关设计思路及具体实现方案。

4.1　机器描述

机器描述相关的模块包括 Archi.v 和 Machregs.v。这两个模块根据目标处理器的体系架构，定义了一些硬件相关的参数。Archi.v 定义了龙芯体系架构相关的信息，例如大小端、64/32 位、对齐方式、非数（NaN）传播等。这些内容可以参考《龙芯架构参考手册》中的相关说明进行配置。比如，龙芯架构只采用小尾端的存储方式，所以设置 big endian：＝false。龙芯浮点数指令非数结果传播有 3 种情况：1）若源操作数中有 SNaN，则选择优先级最高的 SNaN，将其传播为对应的 NaN；2）若源操作数中没有 SNaN 但有 QNaN，则选择优先级最高的 QNaN 作为这条指令的结果；3）除了上面两种情况外，其他需要产生 QNaN 结果的情况都将直接置为缺省的 QNaN 值。参见如下代码片段（可体现 NaN 的传播方式）：

```
(* * Choose the first signaling NaN, if any;
   otherwise choose the first NaN;
   otherwise use default. * )
Definition choose_nan (is_signaling: positive -> bool)
                      (default: bool * positive)
                      (l0: list(bool * positive)):
                      bool * positive :=
  let fix choose_snan(l1: list(bool * positive)) :=
    match l1 with
      | nil =>
        match l0 with nil => default | n :: => n end
      | ((s,p) as n) :: l1 =>
        if is_signaling p then n else choose_snan l1
    end
  in choose_snan l0.
```

Machregs.v 定义机器寄存器，供分配寄存器时使用。这里定义了可用于分配的整数、浮点寄存器，并指明其中可能在某些过程中被使用的寄存器。

定义寄存器时，一方面要遵守龙芯架构本身的物理寄存器实现，另一方面也要根据一些开发上的规范，避开具有固定用途的寄存器，例如龙芯寄存器规范中零寄存器 R0、返回地址寄存器 R1、线程寄存器 R2、栈寄存器 R3、保留寄存器 R21，以及后续翻译过程中的临时寄存器 R20 和 R22，这些寄存器不会在 Machregs.v 中被定义，因而不会被分配。

Machregs.v 中的以 destroyed、temp_for、mregs_for 开头的函数定义了可能在某些过程中用到的寄存器，比如调用函数、定义跳转表时可能会直接使用某些寄存器（而不进行任何的保存和恢复操作），那么这些寄存器就会在这些过程前后被特殊处理。相关的定义主要会遵守后续的实现，比如 TargetPrinter.ml 中定义跳转表相关的指令时用 R12 作为临时寄存器，这里便有如下定义：

Definition destroyed_by_jumptable: list mreg := R12 :: nil.

按照龙芯架构的寄存器约定，被调用者保存寄存器 R22 会被用于保存帧指针[20]。但是，按照 CompCert 的实现，帧寄存器不能为被调用者保存的寄存器。这里当然会向 CompCert 的具体实现妥协，选择 R19（调用者保存寄存器）来保存帧指针。

4.2　调用约定

Conventions1.v 和 Stacklayout.v 描述了特定机器 ABI 的函数调用约定，前者确定了寄存器的使用规范，后者确定了活动记录的具体布局。虽然它们也与特定架构相关，但与 Archi.v 和 Machregs.v 的内容相比，Conventions1.v 和 Stacklayout.v 更多的是参考龙芯 ABI 规范文档，从软性的规范、约定出发做出的定义。

在 Machregs.v 定义的机器寄存器的基础上，Conventions1.v 首先对各寄存器的功能进行分类：被调用者保存的整数/浮点寄存器、调用者保存的整数/浮点寄存器等。之后，Conventions1.v 定义了函数间传递的参数、返回值的位置。

Stacklayout.v 定义了活动记录的结构信息。龙芯文档中对此并没有严格的定义，所以这里参考了 RISC-V 版的具体实现。从栈底到栈顶，栈帧的组成部分分别是：函数参数、之前函数的栈帧、函数返回地址、调用者保存的寄存器的值、局部栈帧单元、Cminor 在栈上分配的数据。

4.3　指令选择

在 Cminor 变换到 CminorSel 时，CompCert 会执行指令选择的工作，所涉及的目标处理器相关模块包括 Op.v，SelectOp.v，SelectLong.v。编写代码时编写的是 SelectOp.vp 和 SelectLong.vp，SelectOp.v 和 SelectLong.v 是通过这两个文件自动生成的。

Op.v 根据目标处理器能在一条指令（或多条）内完成的运算，定义了条件分支的布尔条件（condition）、算术与逻辑运算（operation）、寻址模式（addressing）的语法与形式化语义。这些定义会在之后的中间语言（CminorSel，RTL，LTL，

Mach)的翻译、证明中用到。

布尔条件包括整型数之间的比较、整型数与立即数的比较、浮点数之间的比较等。

运算符包括加法这样的数值运算符、取反这样的逻辑运算符、整型数与立即数的转换等。这些运算符大多与龙芯中本身存在的汇编指令一一对应,例如 Oadd 与汇编 add. w 对应。但为了中间处理方便,这里也有一些例外,比如定义了 Onot 运算符,而龙芯指令中并没有 not 指令,但可以用 nor R0 实现;定义 Ocmp(condition)用于计算布尔条件表达式的值。这些运算符会在 Asmgen. v 的 transl_op 函数中得到进一步翻译,大多数运算符都可以翻译成一条汇编指令,少数翻译成多条,Ocmp(condition)则会调用其他更多的辅助函数细化情况分别进行处理。

对于涉及整型数的指令,运算符定义中往往为 32 位整型数和 64 位整型数分别定义,例如加法指令 Oadd 和 Oaddl,其中后缀 l 表示长(long)整型数。龙芯架构确实为 32 位整型数和 64 位整型数分别提供了加法指令 add. w 和 add. d,但对于龙芯指令中字长无关的与运算指令 and,CompCert 中仍然需要分别定义指令 Oand 和 Oandl。这主要是由 CompCert 本身形式化证明的特点决定的。为实现形式化证明,CompCert 需要定义中间语言的形式化操作语义。在 Op. v 的 eval_operation 函数中有如下定义:

```
Oadd,v1::v2::nil=> Some(Val.add v1 v2)
Oaddl,v1::v2::nil=> Some(Val.addl v1 v2)
```

CompCert 的 Val 库为不同位数的整型数提供的计算方法不同,因此这二者的形式化语义不同,所以需要分别定义。当然,最终输出龙芯汇编的时候还是均输出 and。

不同于汇编语言中的寻址模式,Op. v 中定义的寻址模式主要是从地址空间的角度说明的,包括寄存器相对寻址、全局变量寻址、栈变量寻址。参见如下代码片段(addressing 代码示例):

```
Inductive addressing:Type:=
  | Aindexed:ptrofs-> addressing
    (** r Address is [r1+offset] *)
  | Aglobal:ident-> ptrofs-> addressing
    (** r Address is global plus offset *)
  | Ainstack:ptrofs-> addressing.
    (** r Address is [stack_pointer+offset] *)
```

在 Op. v 提供的定义的基础上,SelectOp. v 和 SelectLong. v 会进行指令的选择,前者处理 32 位的情况,后者处理 64 位的情况。下面仅以 SelectOp. v 为例进行说明。backend/Selection. v 会调用 SelectOp. v 中的函数(如 add 函数),对 Cminor 语言进行翻译,所以 SelectOp. v 需要提供对应的函数接口,借助 Op. v 中定义的操作符,完成功能上的表达。在直接翻译之外,SelectOp. v 还会识别表达式的组合,选用更合适的指令。例如,对于与立即数指令,立即数为 0 时,表达式结果为常数 0;立即数为 −1(比特位全 1)时,结果即为另一源表达式的结果;否则,还可以考虑使用结合律对表达式进行优化。参见下列代码片段(andimm 代码示例):

```
Nondetfunction andimm(n1:int)(e2:expr):=
  if Int.eq n1 Int.zero then Eop(Ointconst Int.zero) Enil
  else if Int.eq n1 Int.mone then e2
  else match e2 with
  | Eop(Ointconst n2) Enil=> Eop(Ointconst(Int.and n1 n2)) Enil
  | Eop (Oandimm n2)(t2::Enil)=> Eop(Oandimm(Int.and n1
      n2))(t2::Enil)
  | _=> Eop(Oandimm n1)(e2::Enil)
  end.
```

4.4 中间代码优化

在 RTL 到 LTL 的变换过程中,CompCert 会做多项优化工作,比如常量传播、冗余代码删除、公共子表达式删除等。涉及的目标处理器相关的模块包括 ValueAOp. v、ConstpropOp. v(由 ConstpropOp. vp 自动生成)、CombineOp. v、NeedOp. v。它们都是在 Op. v 定义的操作符的基础上进行分析优化的。

ValueAOp. v 对操作符进行静态分析,供之后的常量传播、公共子表达式消除阶段使用。

ConstpropOp. v 对操作符进行长度缩减。RTL 中间语言已经可以进行数据流分析,一些表达式中的参数是静态已知的,可以对一些特殊情况进行优化。例如,乘立即数时,如果立即数是 0,1 或 2 的倍数,那么就可以用他它指令代替乘法指令。参见下列代码片段(make_mulimm 代码示例):

```
Definition make_mulimm(n:int)(r1 r2:reg):=
  if Int.eq n Int.zero then
    (Ointconst Int.zero,nil)
  else if Int.eq n Int.one then
    (Omove,r1::nil)
  else
    match Int.is_power2 n with
    | Some l=>(Oshlimm l,r1::nil)
    | None=>(Omul,r1::r2::nil)
    end.
```

CombineOp. v 对操作符的组合进行等价替换(例如连续加两个立即数等于直接加这两个立即数的和),供之后的公共子表达式消除阶段使用。参见下列代码片段(combine_op 代码示例):

```
Function combine_op (op:operation)(args:list valnum):
                         option(operation * list valnum):=
  match op,args with
  | Oaddimm n,x::nil=>
    match get x with
    | Some(Op(Oaddimm m) ys)=> Some(Oaddimm
      (Int.add m n),ys)
    |=> None
    end
```

NeedOp. v 对 Op. v 中定义的操作符进行需要性分析,供之后的冗余代码删除阶段使用。

4.5 生成目标汇编

生成目标汇编相关的模块包括 Asm. v, Asmgen. v, Asmexpand. ml, TargetPrinter. ml。Asm. v 定义汇编语言,Asmgen. v 将 Mach 语言翻译为汇编,Asmexpand. ml 对一些内置函数、伪指令做展开,TargetPrinter. ml 做最终的格式化输出。

4.5.1 汇编指令定义

Asm. v 主要定义了寄存器和汇编指令,同时也定义了

机器寄存器的翻译和指令执行的形式化语法等。

这里定义的寄存器与 Machregs. v 中定义的机器寄存器不同。Machregs. v 中定义的机器寄存器是为了分配给 RTL 中的伪寄存器的，只需要定义可分配的整数寄存器和浮点寄存器。Asm. v 中定义的寄存器是与后端架构相关的，即需要定义汇编指令中所有会用到的寄存器(包括状态寄存器等)。preg_of 函数会定义机器寄存器向汇编寄存器的翻译方式。

指令的选择一方面要参照相关的指令手册(《龙芯架构参考手册》)，另一方面也要根据 Asmgen. v 中翻译 Mach 语言的需求，适当地舍弃或特殊处理某些指令。同时，这里也会定义一些伪指令，方便中间处理。与 Op. v 中的定义同理，为了实现形式化证明，这里也会为龙芯的部分 32 位与 64 位通用指令(and 等)提供分别供 32 位使用和供 64 位使用的两个定义。下面介绍一些特殊处理的指令和伪指令。

龙芯架构用 fcmp. cond. {s/d}来进行浮点比较，其中的 cond 有 22 种，分别对应不同的比较条件。本次实现中，只取了 cond 为 eq，lt，le 的 3 种情况，即浮点相等、浮点小于、浮点不大于的比较，并分别设计对应的指令和语法。

龙芯架构的浮点转换指令也比较复杂。以 float 到 int 的转换为例，龙芯需要使用 ftintrz. w. s fd，fs 和 movfr2gr. s rd，fs 两条指令，前者将浮点寄存器 fs 中的单精度浮点数转换为整数型定点数，得到的整数型定点数以“向零方向舍入”的方式写入到浮点寄存器 fd 中，后者将浮点寄存器 fs 低 32 位的值符号扩展后写入通用寄存器 rd 中。如果直接按照龙芯的样式设计两条指令，可能会在中间进行形式化定义与形式化证明的过程中带来许多不必要的麻烦。考虑到这两条指令往往都是同时使用的，这里可以定义只定义一条指令 Pfcvtws 用于 float 到 int 的转换，仅在最后格式化输出时输出实际的两条指令。由于这两条指令中间会用到寄存器 F0，所以还需要在 Machregs. v 的相关函数中添加定义。参见下列代码片段(destroyed_by_op 代码示例)：

```
Definition destroyed_by_op(op:operation):list mreg:=
  match op with
  |Ointoffloat|Ofloatofint|Ointofsingle|Osingleofint
  |Olongoffloat|Ofloatoflong|Olongofsingle|Osingleoflong
    => F0::nil
  | _=> nil
  end.
```

按照 CompCert 的设计习惯，Asm. v 中会定义一些伪指令，比如栈帧分配、栈帧释放、标签、加载立即数等。部分伪指令会在 Asmexpand. ml 中被翻译为其他汇编指令序列(再通过 TargetPrinter. ml 格式化输出)，部分伪指令会直接在 TargetPrinter. ml 里被输出为实际的指令。

在本文完成时，龙芯架构参考手册中还没有列出汇编语言中的伪指令，但很多伪指令都是实际存在并可以使用的，例如 move rd，rs、la rd，symbol 等。这里是通过研究 GCC 的翻译表现得出这些指令特性的。

4.5.2 Mach 翻译到 Asm

Asmgen. v 以 transf_program 为起点，不断调用各辅助函数对程序子部分进行翻译，直到完成 Mach 到 Asm 的翻译。此外，Asmexpand. ml 会对一些内置函数、伪指令做展开，并修正一些函数调用时的寄存器位置。不同的后端架构在这里的翻译方式其实大都比较相似。下面仅介绍 Asmgen. v 中对龙芯指令立即数的相关处理方式。

立即数的处理方式相对复杂。C 语言中对整型数的限制是 32 位(长整型数是 64 位)，但在汇编语言中，一条指令往往不能直接操作 32 位或 64 位的立即数。例如，龙芯架构中的 32 位加立即数指令 addi. w rd，rj，si12，其立即数只支持 12 位符号整数的范围，如果需要操作一个更大范围的立即数，则需要使用另外的方法，比如先将立即数写入通用寄存器中，再用 add. w rd，rj，rk 指令做寄存器间的加法。CompCert 的其他部分并不会考虑立即数范围，因此需要在 Asmgen. v 中，对立即数相关的指令做处理。

龙芯架构处理 32 位整型数相关的指令有 3 条：1)加载立即数指令 addi. w rd，rj，si12；2)加载立即数指令 ori rd，rj，ui12；3)加载立即数到高位指令 lu12i. w rd，si20。对于 12 位符号整数的范围内(即不小于 −2048 且不大于 2047)的数，直接用 addi. w rd，rj，si12 指令；对于 12 位符号整数的范围外，但在 12 位无符号整数的范围内(不小于 0 且不大于 4095)的数，即大于 2047 不大于 4095 的数，直接用 ori rd，rj，ui12 指令；如果这个数无法用 12 位整数表示，则用 lu12i. w rd，si20 和 ori rd，rj，ui12 的组合进行处理，前者加载高 20 位，后者加载低 12 位。

对 64 位整型数，其低 32 位的处理方式与 32 位整型数的类似，其高 13 到 32 位用 lu32i. d rd，si20 指令加载，高 12 位用 lu52i. d rd，rj，si12 指令加载。以下列举了一些从 C 语言翻译到龙芯汇编指令的例子(处理立即数的汇编指令)：

```
f:
  # a+=1
  addi.w $r4,$r4,1
  # b+=2048
  ori $r12,$r0,2048
  add.w $r5,$r5,$r12
  # c+=0x12345678
  lu12i.w $r12,0x12345
  ori $r12,$r12,0x678
  add.w $r6,$r6,$r12
  # d+=0x123456789abcdef0
  lu12i.w $r12,0xfffffffff9abcd
  ori $r12,$r12,0xef0
  lu32i.d $r12,0x45678
  lu52i.d $r12,$r12,0x123
  add.w $r7,$r7,$r12
```

定义好立即数的加载方式后，就可以比较方便地处理 addi 这样的操作寄存器与操作数的指令了。如果操作数比较大(大于 12 位符号整数的范围)，就先用上面提到的方法，把立即数写入一个临时寄存器中，再用对应的 add 这样的操作寄存器与寄存器的指令进行操作。除了 addi 外，slti 和 sltui 也是一样的处理方式。不过 andi，ori，xori 的处理略有不同，因为它们的操作数的范围是 12 位无符号整数而不是 12 位符号整数，但理念都是一样的。具体的操作方式，可以参考如下代码(andimm32，orimm32，xorimm32 的处理方式)：

```
Definition opui32(op:ireg-> ireg0-> ireg0-> instruction)
  (opimm:ireg-> ireg0-> int-> instruction)
  (rd rs:ireg)(n:int)(k:code):=
```

```
  if Int.eq n(Int.zero_ext 12 n) then
    opimm rd rs n::k
  else
    loadimm32 R20 n(op rd rs R20::k).
Definition andimm32:= opui32 Pandw Pandiw.
Definition orimm32:= opui32 Porw Poriw.
Definition xorimm32:= opui32 Pxorw Pxoriw.
```

4.5.3 Asm 文本输出

TargetPrinter.ml 定义了格式化输出函数，除了翻译 Asm.v 中定义的指令外，还会翻译汇编中的段信息。

4.6 可变参数

C 语言的 stdarg.h 库提供了可变参数的定义方法，其中包括函数 va_arg 的实现。CompCert 会在 runtime/$(arch)/vararg.S 文件中提供用汇编实现的函数__compcert_va_int32，__compcert_va_int64，__compcert_va_float64 和__compcert_va_composite，分别对应 va_arg(v,l)中 l 的类型为 int，long，double 和 struct 的情况。出于方便，vararg.S 会在 sysdeps.h 头文件中定义一些架构相关的宏。参见如下代码示例：

```
FUNCTION(__compcert_va_int32) # a0=ap parameter
  ldptr $t0,$a0,0 # t0=pointer to next argument
  addi $t0,$t0,WORDSIZE # advance ap
  stptr $t0,$a0,0 # update ap
  ld.w $a0,$t0,-WORDSIZE # load it and return it in a0
  jr $ra
ENDFUNCTION(__compcert_va_int32)
```

4.7 配置文件

Configure，cparser/Machine.ml，cparser/Machine.mli，driver/Configuration.ml，driver/Frontend.ml 文件中对不同处理器架构进行了不同的配置。在重定向到龙芯处理器时，需要在这些文件中增加 loongarch 相关的配置。

4.8 测试相关

CompCert 编译器的代码里 test/文件夹下提供了测试用例，也准备了相关的 Makefile 和脚本文件供正确性、性能的测试。在编译出编译器的可执行文件后，运行测试前，还需要修改几个文件的内容：修改 test/endian.h，增加大小端的配置；修改 test/regression/extasm.c，增加 64 位的宏定义；增加 test/regression/builtinsloongarch.c 文件，为架构相关的内置函数提供测试用例。

在本文完成时，龙芯的安装源中没有 ocamlopt，只有版本为 4.05.0 的 ocamlc，这影响到了 abi 的测试：abi 的相关测例中有使用到 4.06 版的 ocamlopt 的相关特性，不能直接使用。为了进行相关的测试，对 test/abi/Makefile 进行了修改，将 ocamlopt 改为 ocamlc，同时修改 test/abi/generator.ml，增加 List.init 的相关函数定义。若是龙芯的安装源中相关的工具版本得以更新，可以省略这一步操作。

在 test/文件夹下或在其子文件夹下运行 make all 对所有 C 源文件进行编译，运行 make test 对正确性进行测试，运行 make bench 对性能进行测试。在使用 GCC 测试时，需要对各子文件夹下的 Makefile 进行修改，把编译器(CC)改为 GCC，并调整编译选项(CFLAGS)。

[1] https://github.com/HuShaoRu/CompCert

5 性能测试

在龙芯环境(具体运行环境软硬件配置如表 1 所列)对 CompCert 编译器的性能进行测试，并与 GCC 不同优化等级(-O0，-O1，-O2，-O3)下的表现作对比。使用的测例涉及信号处理、物理模拟、3d 图像、密码学、文本压缩等方面应用。测试的操作方法可见 4.8 节。

表 1 测试环境

Table 1 Test environment

类型	参数
处理器	Loonson-3A5000HV
操作系统	Loonnix GNU/Linux 20(DaoXiangHu)
内核	Linux 4.19.167-rc5.lnd.l-loongson-3(loongarc64)
GCC 版本	8.3.0
Coq 版本	8.9.0
Ocaml 版本	4.05.0
Menhir 版本	20190626

表 2 列出了在不同的源代码条件下，GCC-O0，CompCert，GCC-O1，GCC-O2，GCC-O3 分别生成的程序的运行时间。

表 2 龙芯处理器上 CompCert 和 GCC 8.3.0 性能对比

Table 2 Performance comparison between CompCert and GCC 8.3.0 on Loongson processor

测试文件	Gcc-O0	CompCert	GCC-O1	GCC-O2	Gcc-O3
fib	0.124	0.092	0.079	0.052	0.052
integer	0.070	0.032	0.020	0.016	0.016
qsort	0.258	0.163	0.149	0.150	0.150
fit	0.162	0.096	0.086	0.082	0.083
fftsp	0.177	0.065	0.060	0.056	0.056
sha1	0.188	0.105	0.071	0.076	0.067
sha3	0.320	0.058	0.026	0.027	0.019
aes	0.317	0.176	0.122	0.108	0.108
almabench	0.588	0.537	0.387	0.383	0.369
lists	0.260	0.097	0.097	0.096	0.097
binarytrees	0.094	0.068	0.067	0.065	0.063
fannkuch	0.885	0.267	0.246	0.214	0.247
knucleotide	0.340	0.178	0.162	0.159	0.154
mandelbrot	0.369	0.189	0.178	0.161	0.160
nbody	0.841	0.265	0.272	0.261	0.219
nsieve	0.389	0.176	0.158	0.158	0.158
nsievebits	0.552	0.174	0.154	0.219	0.217
special	0.508	0.384	0.383	0.192	0.192
vmach	0.289	0.104	0.077	0.077	0.077
bisect	0.174	0.143	0.133	0.138	0.138
chomp	0.527	0.219	0.223	0.182	0.182
perlin	0.245	0.111	0.057	0.036	0.037
siphash24	0.562	0.190	0.112	0.068	0.050
arcode	0.449	0.240	0.198	0.183	0.184
lzw	29.908	2.572	2.586	2.524	2.536
lzss	3.111	1.933	1.762	1.555	1.480
raytracer	3.099	1.397	1.233	1.151	1.154
spass	40.609	11.063	8.266	8.067	8.035

平均说来，CompCert 生成的程序的运行速度比 GCC-O0 快 166%，比 GCC-O1 慢 16%，比 GCC-O2 慢 23%，比 GCC-O3 慢 25%。可以认为 CompCert 编译器在龙芯处理器后端有比较好的性能表现。

相较于其他已完整实现的后端，CompCert 与 GCC 的表现差异也与这里比较接近：例如，PowerPC 后端的 CompCert 生成的程序的运行速度，大约比 GCC-O0 快 100%，比 GCC-O1 慢 10%，比 GCC-O2 慢 15%，比 GCC-O3 慢 20%[21]。相较而言，可以认为本文 CompCert 编译器的龙芯处理器重定

向已经取得了比较好的效果。

结束语 CompCert 是国际上著名的 C 语言高可信编译器，也是编译器形式化验证工作的杰出代表，在学术界和工业界都具有举足轻重的地位，在安全攸关领域的影响力巨大。

本文介绍了基于 CompCert(版本 3.10)实现的面向龙芯处理器的 C 语言可信编译器项目[22]，后者是将 CompCert 编译器重定向到国产处理器架构的首次尝试，它在 CompCert 的原有框架上继续开发，增加了龙芯架构后端的相关文件项，使得 CompCert 编译器可以将 C 源代码编译成龙芯体系架构的汇编语言代码。所实现的编译器支持与 CompCert 兼容的 C 语言特性(可以通过 CompCert 编译器源代码中 test/文件夹下提供的全部测试用例1))，其生成的代码的性能接近 GCC-O1 的效果，可满足很多场景的使用需求。目前相关代码已经在 GitHub 上开源2)。

除重定向功能外，该项目[22]同时完成了大部分正确性证明相关模块的移植工作，后续项目[23]已涵盖了所遗留的少数定理证明(已合并至 GitHub 源代码2))。限于篇幅，本文未过多涉及正确性证明，相关内容的介绍将另外成文。

后续可以在以下几个方面开展进一步工作：1)测试该编译器在 loongarch32 架构上的兼容性；2)增加更多的指令，对一些表达式作进一步优化，或者增加更多的编译器内置函数；3)等待龙芯手册对龙芯架构做更详细的介绍，对部分实现细节进行调整。

参 考 文 献

[1] YANG X J, CHEN Y, ERIC E, et al. Finding and understanding bugs in C compilers[C]// Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI 2011). 2011:283-294.

[2] ZHANG Q R, SUN C N, SU Z D. Skeletal Program Enumeration for Rigorous Compiler Testing[C]// Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI 2017). 2017.

[3] MCCARTHY JPAINTER J. Correctness of a compiler for arithmetical expressions[C]// Mathematical Aspects of Computer Science, volume 19 of Proceedings of Symposia in Applied Mathematics. 1967:33-41.

[4] MILNER R, WEYHRAUCH R. Proving compiler correctness in a mechanized logic[C]// Proceedings of 7th Annual Machine Intelligence Workshop, volume 7 of Machine Intelligence. 1972:51-72.

[5] DAVE M A. Compiler verification: a bibliography [J]. ACM SIGSOFT Software Engineering Notes, 2003, 28(6):2-2.

[6] LEROY X. Formal verification of a realistic compiler[C]// Communications of The ACM. 2009:107-115.

[7] LEROY X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant[C]// Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2006:42-54.

[8] PNUELI A, SIEGEL M, SINGERMAN E. Translation Validation[C]// Proceedings of TACAS'98, Lecture Notes in Computer Science. 1998:151-166.

[9] KÄSTNER D, LEROY X, BLAZY S, et al. Closing the gap—the formally verified optimizing compiler CompCert[C]// Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium(SSS'17). 2017:163-180.

[10] LEROY X, BLAZY S. Formal verification of a C-like memory model and its uses for verifying program transformations[J]. Journal of Automated Reasoning, 2008, 41(1):1-31.

[11] YANG P, WANG S Y. Analysis of Target Code Generation Mechanism of CompCert Compiler[J]. Computer Science, 2020, 47(9):17-23.

[12] LEROY X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4):363-446.

[13] GEORGE L, APPEL A W. Iterated register coalescing[J]. ACM Trans. Prog. Lang. Syst., 1996, 18(3):300-324.

[14] RIDEAU S, LEROY X. Validating register allocation and spilling[C]// CC 2010. 2010:224-243.

[15] BLAZY S, ROBILLARD B, APPEL A W. Formal verification of coalescing graph-coloring register allocation[C]// ESOP 2010. 2010:145-164.

[16] 龙芯中科. 从必然王国到自由王国，龙芯重磅推出自主指令系统架构 LoongArch [EB/OL]. [2021-05-10]. https://www.loongson.cn/newsDetails/1913.

[17] 龙芯中科. 重磅发布|基于龙芯架构的新一代处理器龙芯 3A5000 正式发布[EB/OL]. [2021-09-03]. https://www.loongson.cn/newsDetails/22.

[18] 龙芯中科技术股份有限公司. 龙芯架构参考手册[M]. 2021.

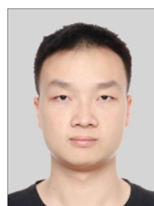
[19] 胡伟武等. 计算机体系结构基础[M]. 机械工业出版社, 2021.

[20] Loongson Technology Corporation Limited. Loongarch elf abi specification [OL]. https://loongson.github.io/LoongArch-Documentation/LoongArch-ELF-ABI-EN.html.

[21] LEROY X, BLAZY S, KÄSTNER D, et al. CompCert—A Formally Verified Optimizing Compiler [C]// ERTS 2016. 2016..

[22] 胡少儒. CompCert 编译器的龙芯处理器重定向[D]. Beijing: Tsinghua University, 2022.

[23] 王隽伟. 机器证明的 CompCert 编译器龙芯处理器后端[D]. Beijing: Tsinghua University, 2023.



HU Shaoru, born in 1999, B.S. His main research interests include formal verification and compiler, and so on.

WANG Shengyuan, born in 1964, Ph.D, associate professor. His main research interests include programming languages and systems, compilers and formal methods.

1) https://compcert.org/

2) http://github.com/HuShaoRu/Complert