计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191100017

基于 LLVM 编译器的节点融合优化方法

胡 浩[1] 沈 莉[2,3] 周清雷[1] 巩令钦[1]
1 郑州大学信息工程学院 郑州 450000
2 中国科学技术大学计算机科学与技术学院 合肥 230000
3 无锡江南计算技术研究所 江苏 无锡 214083

摘 要 LLVM 是以 C++编写的架构编译器的框架系统,支持多后端和交叉编译,用于优化程序的编译时间、链接时间、运行时间和空闲时间。节点融合是一种简单有效的优化方法,其基本思想为将多个节点优化为一个高效的融合节点,减少诸如指令、寄存器、时钟周期和访存等开销,以达到减少程序运行时间,提升访存效率等目的。为了提升 LLVM 编译器的性能,文中在 LLVM 编译流程的中间表示阶段和 DAG 合并阶段、指令选择阶段提出了节点融合优化方法。在国产平台申威处理器下,以 CLANG 和 FLANG 为编译器前端,LLVM 为编译器后端,基于 SPEC CPU2006 测试集进行了评估,实验结果表明,节点融合优化有利于提高编译器性能和减少程序运行时间,优化后最大加速比为 1.59,平均加速比为 1.13。

关键词:LLVM;节点融合;中间表示;DAG 合并;指令选择;国产平台
中图法分类号 TP311

Node Fusion Optimization Method Based on LLVM Compiler

HU Hao[1],SHEN Li[2,3],ZHOU Qing-lei[1] and GONG Ling-qin[1]
1 School of Information Engineering,Zhengzhou University,Zhengzhou 450000,China
2 School of Computer Science and Technology,University of Science and Technology of China,Hefei 230000,China
3 Wuxi Jiangnan Computer Technology Research Institute,Wuxi,Jiangsu 214083,China

Abstract LLVM is a framework system of architecture compiler written in C++,which is a cross-compiler that supports multiple back-ends. It can optimize program compilation time,link time,run time,and idle time. Node fusion is a simple and effective optimization method. The basic idea is to optimize multiple nodes into an efficient fusion node. This optimization can reduce overhead such as instructions,registers,clock cycles,memory access,so as to reduce program running time and improve memory access efficiency. In order to improve the performance of the LLVM compiler,node fusion optimization algorithm is proposed for the LLVM compiler in the intermediate presentation phase,DAG combine phase and instruction selection phase. Under the domestic platform Sunway processor,with CLANG and FLANG as the front end and LLVM as the back end of the compiler,LLVM is evaluated based on the SPEC CPU2006 test set. The results show that node fusion optimization is beneficial to improve compiler performance and reduce program running time. The optimized maximum speedup ratio is 1.59 and the average speedup ratio is 1.13.

Keywords LLVM,Node fusion,Intermediate representation,DAG combine,Instruction selection,Domestic platform

1 引言

LLVM[1]是底层虚拟机(Low Level Virtual Machine)的简称,它是以伊利诺伊大学厄巴纳-香槟分校开源许可(University of Illinois/NCSA Open Source License)进行开源的框架编译器。LLVM 是一种支持多后端的编译器[2],输入为中间表示 LLVM IR[3](Intermediate Representation),输出为给定平台后端的可执行代码,即同一个中间表示可生成不同平台后端的可执行代码。LLVM 提供了平台无关的优化工具 OPT,使用该工具可对中间表示 LLVM IR 进行多种平台无关的优化[4],在中间表示降级到可执行代码的过程中也可进行平台无关和平台相关的多种优化。节点融合是一种简单有效的优化方法,在 LLVM 编译的多个阶段都可以实现节点融合优化,根据实现方式和阶段的不同,可分为平台无关和平台相关的节点融合优化。节点融合优化的思想与窥孔优化(Peephole optimization)[5]类似,都是一种局部的优化方法。与激进的窥孔优化不同,节点融合优化通过对融合前后的节点进行代价评估,根据代价的大小决定是否进行节点融合,避免过度优化。本文针对 LLVM 编译器在中间表示阶段、有向无环图(Directed Acyclic Graph,DAG)合并阶段以及指令选

本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家自然科学基金(61572444)
This work was supported by the National Natural Science Foundation of China (61572444).
通信作者:胡浩(witstorm@163.com)

择阶段的节点融合优化进行了研究并在国产申威平台上进行了实现。本文基于 SPEC CPU2006 测试集[6]进行评估，实验结果表明，节点融合优化有利于提高编译器性能、减少程序的运行时间，优化后最大加速比为 1.59，平均加速比为 1.13。

2 研究背景

2.1 LLVM 框架

LLVM 是以面向对象编程语言 C++编写的构架编译器的框架系统，是一种支持多后端的交叉编译器，可对程序的编译时间、链接时间、运行时间和空闲时间进行优化[7]。LLVM 编译器基于传统的三段式设计，通过翻译成通用的中间表示语言作为中端优化器的输入，用于支持不同的前端语言和架构[8]，其结构如图 1 所示。

图 1 LLVM 结构

Fig. 1 Structure of LLVM

从图 1 中可以看出，LLVM 的编译流程可以分为三大部分：高级语言前端、中间代码优化器(通用优化器)和后端代码生成器。高级语言前端将使用高级语言 C，Fortran，Java 等编写的程序转换为 LLVM IR，与高级语言前端和目标平台处理器后端均独立的中间代码优化器则对转换得到的 LLVM IR 进行优化，经过优化后的 IR 通过后端代码生成器生成针对目标平台处理器的机器代码。

2.2 中间表示

LLVM IR 属于高级语言前端的输出，将作为中间代码优化器的输入。中间表示应该具有两个重要的性质：1)易于产生；2)易于翻译成目标平台代码[9]。易于产生可以保证不同的高级语言源程序易于转换为中间表示，也可以与其他中间表示相互转换。易于翻译成目标平台代码表明中间表示应具有高度抽象的特性。中间代码优化器对中间表示进行优化时，其输入是 LLVM IR，其输出也是 LLVM IR，所以对中间表示的优化不会影响其翻译成目标平台代码。LLVM IR 有 3 种表现形式：1)编译过程中内存形式的中间表示；2)磁盘中的比特码(bitcode)；3)用户可读的汇编码。其中，编译过程中内存形式的中间表示存在于各个优化和分析 Pass 中，比特码和汇编码分别以.bc 格式和.ll 格式保存在磁盘中。LLVM 编译器中 LLVM IR 存在的阶段如图 2 所示。

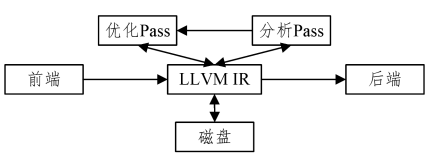

图 2 LLVM IR 存在阶段

Fig. 2 Phases diagram of LLVM IR

从图 2 中不难看出，LLVM IR 是连接前端和后端的重要枢纽。前端将高级程序语言转化为平台无关的中间表示，中端对平台无关的中间表示进行分析、优化，后端将中间表示降级为目标平台的可执行文件或目标平台文件。LLVM IR 所处的位置决定了其应是平台无关的语言并且易于产生和翻译成目标平台代码。

2.3 中间表示降级

DAG 是一种重要的数据结构，指的是一个无回路的有向图，在寻求最短路径、数据压缩等多种算法中均有使用。LLVM 在对中间表示进行降级的过程中，会将 LLVM IR 转换成 DAG 的形式，然后对 DAG 图进行降级操作，其处理流程如图 3 所示。

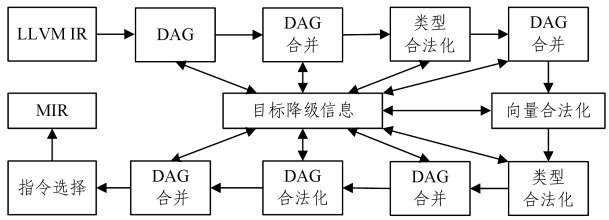

图 3 LLVM IR 降级流程

Fig. 3 Lowering processes of LLVM IR

从图 3 中可以看到，在初始阶段 LLVM IR 是平台无关的，LLVM IR 转化成 DAG 图时，需要目标平台降级信息(TargetLoweringInfo，TLI)，然后再进行 DAG 合并、类型合法化和 DAG 合法化等处理，最后再进行指令选择处理生成 MIR(Machine IR)，完成 LLVM IR 的降级。在整个降级的过程中主要是 DAG 合并和合法化这两个过程交替进行，逐步完成中间表示的降级操作。

3 节点融合

3.1 节点融合简介

节点融合指将多个节点融合为一个高效节点，节点融合优化方法在许多编译器中均有应用。比如，加速线性代数编译器(Accelerated Linear Algebra，XLA)利用节点融合优化将计算图中的多个算子融合为一个高效算子，以提高执行速度，改善内存使用，减少对自定义操作的依赖。节点融合过程如图 4 所示。

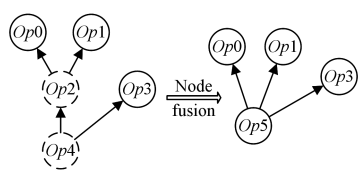

图 4 节点融合

Fig. 4 Node fusion

图 4 中的圆圈为操作节点，圆圈中的标识符为节点名称，虚线圆圈为待融合的操作节点，箭头为节点间的数据依赖关系。图 4 左图中，操作节点 $Op2$ 依赖操作节点 $Op0$ 和操作节点 $Op1$ 的运算结果，操作节点 $Op4$ 依赖操作节点 $Op2$ 和操作节点 $Op3$ 的运算结果。图 4 右图中，操作节点 $Op5$ 依赖操作节点 $Op0$，$Op1$ 和 $Op3$ 的运算结果。图 4 左图经过节点融合优化，将操作节点 $Op2$ 和 $Op4$ 融合成右图中的操作节点 $Op5$，同时维持了 $Op2$ 和 $Op4$ 的数据依赖关系。未进行节点融合前完成图 4 左图所有的操作需要 5 个操作，进行节点融合后需要进行的操作只需要 4 个，数据依赖链的长度由 2 变为 1。

中间表示中的操作或节点在降级处理过程中降级为目标平台指令集中指令，所以可使用指令在目标平台上运行时所

花费的时钟周期数作为代价，对节点融合前后的代价进行评估。用 $cost(OpN)$ 表示 OpN 操作降级为目标平台指令集中的指令后执行指令所花费时钟周期数，则图 4 左图的总代价 costBefore(融合前的总代价)为 $cost(Op0)+cost(Op1)+cost(Op2)+cost(Op3)+cost(Op4)$，右图的总代价 costAfter(融合后的总代价)为 $cost(Op0)+cost(Op1)+cost(Op3)+cost(Op5)$，其中 $Op5$ 是由节点 $Op2$ 和 $Op4$ 融合而来，节点 $Op5$ 的代价满足不等式 $\max(cost(Op2),cost(Op4)) <= cost(Op5) < cost(Op2)+cost(Op4)$，即融合后节点所需的时钟周期数小于节点融合前所需的总时钟周期数，同时融合后节点所需的时钟周期数不小于融合前节点所需的时钟周期数中的最大值。$costBefore-costAfter=cost(Op4)+cost(Op2)-cost(Op5)$，从而可知 $costBefore>costAfter$，即进行节点融合后其所花费的代价小于节点融合前的代价，所以可以进行节点融合优化。图 4 描述的是较理想情况下的节点融合优化，需要指出的是：并非在任何情况下，节点融合后的代价都小于节点融合前的代价，如图 5 所示。

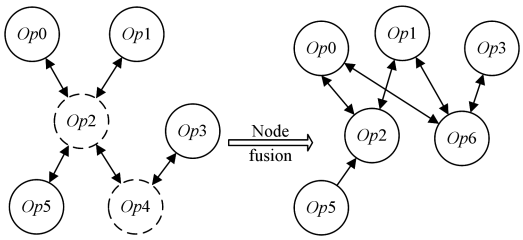

图 5　不合适的节点融合

Fig. 5　Inappropriate node fusion

图 5 左图中的 $Op2$ 和 $Op4$ 是需要进行节点融合的两个节点，融合后转化为右图中的 $Op6$ 节点。图 5 与图 4 不同的是节点 $Op2$ 多了一个使用者节点 $Op5$，该节点的操作依赖 $Op2$ 操作的结果，所以进行节点融合后，$Op2$ 仍将保留。图 5 左图的代价 $costBefore$(融合前的总代价)为 $cost(Op0)+cost(Op1)+cost(Op2)+cost(Op3)+cost(Op4)+cost(Op5)$，右图的代价 $costAfter$(融合后的总代价)为 $cost(Op0)+cost(Op1)+cost(Op3)+cost(Op2)+cost(Op6)+cost(Op5)$，其中 $\max(cost(Op2),cost(Op4)) <= cost(Op6) < cost(Op2)+cost(Op4)$。$costBefore-costAfter=cost(Op4)-cost(Op6)$，从而 $costBefore-costAfter$ 小于或等于 0，即进行节点融合后其代价不小于节点融合前的代价，进行节点融合后不能带来性能的提升，所以不宜进行节点融合优化。

3.2　LLVM 中的节点融合

节点融合优化在 LLVM 编译流程的各个阶段都可能会发生，本文选取中间表示阶段、DAG 合并阶段以及指令选择阶段进行了研究和实现。这 3 个阶段都可以对节点进行融合优化处理，不同阶段处理的效果和方式各有差异，以下将进行各阶段节点融合可行性分析。

中间表示阶段。静态单赋值[10](Static Single-Assignment，SSA)形式的中间表示具有显式的 use-def 链，通过 use-def 链可以得到变量之间的数据流关系，利用数据流关系可以进行节点融合优化。中间表示阶段的节点融合优化是将匹配上的多条中间表示语句通过内建函数的方式发射到 LLVM IR 中，替换原先的中间表示语句。LLVM IR 应具有的一个属性是平台无关，所以在中间表示阶段进行的节点融合属于平台无关优化。当然进行平台相关的优化也是可行的，但这势必会破坏中间表示应具有的平台无关的属性，导致生成的中间表示仅适合于特定的平台后端，同时也会影响其他平台无关优化，比如常量传播[11]和公共子表达式消除[12]等。此外，在中间表示层进行的平台相关的节点融合优化生成的融合节点如果不被平台后端所支持，则该融合节点将在 DAG 降级阶段被拆分为多个节点，导致此节点融合优化失效，反而增加了编译时间，不利于编译器性能提升。

DAG 合并阶段。DAG 合并阶段由于可以获取到 TLI 信息，因此既可以进行平台相关的节点融合优化，又可以进行平台无关的节点融合优化。该阶段在保留了前端丰富的中间表示信息的基础上，结合后端平台指令集的信息，进行针对性的节点融合优化。在此阶段，由于拥有目标平台后端的指令集信息和架构相关的信息，因此可以保证融合后的节点转化为相应的指令。不仅如此，在该阶段添加的融合优化代码只是针对目标平台后端的修改，不会影响到其他后端，也不会影响 LLVM 的版本升级。此外，在中间表示降级过程中会多次进行 DAG 合并处理，所以在 DAG 合并阶段的节点融合优化也会多次进行。

指令选择阶段。指令选择阶段完成的功能是将所有的平台无关的节点降级为平台相关的节点，经过指令选择阶段后 DAG 只能包含平台相关的节点，否则将导致编译器报错。指令选择阶段的节点融合优化是根据后端指令集描述文件中定义的 Pattern[13]进行模板匹配来实现的。通过 Pattern 匹配的方式进行节点的融合优化，利用了通用的 Pattern 匹配流程，所以实现起来极其简单，只需在后端指令集描述文件中编写节点融合 Pattern 即可。但是 Pattern 匹配的方式无法获取各个节点之间完整的数据流关系，导致无法进行节点融合前后的代价评估，只能尽最大努力进行节点融合，所以可能会导致优化后的代码质量下降。

4　节点融合实现

4.1　中间表示阶段节点融合

中间表示阶段的节点融合指在生成 LLVM IR 或者对 LLVM IR 进行优化的过程中进行的节点融合操作，其输入为 LLVM IR，输出也为 LLVM IR。SSA 形式的 LLVM IR 要求每个变量只被分配一次，并且每个变量在使用之前需被定义。在 SSA 形式的 IR 语句中，其输入操作数为使用点(use 点)，结果值为定义点(def 点)。SSA 形式的这种特点，使得其变量的 use-def 链是显式的，所以通过 IR 语句的操作数可以找到其定义点。以表达式 a×b+c 的中间表示代码为例说明 SSA 形式中间表示的特点，如算法 1 所示。

算法 1　表达式 a×b+c 的 IR 代码

```
1. %0=load double,double* @a,align 8
2. %1=load double,double* @b,align 8
3. %mul=fmul double %0,%1
4. %2=load double,double* @c,align 8
5. %add=fadd double %mul,%2
```

算法 1 中，以%开头的变量都是虚拟寄存器属于局部变量，而@开头的变量为全局变量，赋值运算符右边的 load，

fmul,fadd 分别为装载操作、浮点乘和浮点加[13]操作。比如第 3 条 IR 语句%mul=fmul double %0,%1 中%mul 为 fmul 操作的结果值,其结果值为 double 类型,由于%mul 位于赋值运算符的左边,所以该处为 def 点。Fmul 操作的两个输入操作数分别为%0,%1,其值为 double 类型,%0,%1 位于赋值运算符的右边,所以该处为%0,%1 的 use 点。根据 SSA 形式 IR 的特点,只需找到输出操作数为%0,%1 的 IR 语句就可以知道%0,%1 的定义点分别为第 1 条 IR 语句%0=load double,double * @a,align 8 和第 2 第 IR 语句%1=load double,double * @b,align 8,其中 align 8 表示该 load 操作需要按 8 字节对齐访问内存。

在中间表示阶段进行节点融合优化相对来说比较容易,因为越靠近编译器前端其保留的信息越丰富,越靠近后端其信息丢失越严重。该阶段的节点融合优化,由于不需要目标平台后端的信息,所以属于平台无关的优化。在中间表示阶段有两处可以进行节点融合优化:一个是在前端将抽象语法树转化为 LLVM IR 时进行节点融合优化;另一个使用 LLVM 提供的平台无关的优化工具 OPT 进行节点融合优化。OPT 是一个通用优化器,其中包含了多个优化 Pass,用来对 LLVM IR 进行平台无关的优化,比如公共子表达式消除(Early CSE)和内联优化(Inline)等。算法 1 中的中间表示进行浮点乘加节点融合优化的如算法 2 所示。

算法 2 中间表示层节点融合

```
1. %0=load double,double * @a,align 8
2. %1=load double,double * @b,align 8
3. %2=load double,double * @c,align 8
4. %3= call double @llvm. fmuladd. f64(double %0,double %1,
   double %2)
```

在算法 2 中,可以看到第 3 条 IR 语句%mul=fmul double %0,%1 和第 5 第 IR 语句%add=fadd double %mul,%2 融合为算法 2 中的第 4 条 IR 语句%3=call double @llvm. fmuladd. f64(double %0,double %1,double %2)形式的内建函数调用,该内建函数会在后续的中间表示降级过程中逐步转化为相应的浮点乘加指令。中间表示阶段的节点融合算法如算法 3 所示。

算法 3 中间表示阶段节点融合

```
输入:(Op)/ * 待融合节点 * /
输出:( fusionOp)/ * 融合后节点 * /
1. costBefore ← cost(Op)
2. costAfter ← 0
3. fusionOp ← Op
4. case Op. getOpcode() in
5. Opcode1 )
6.     Op0←Op. getOperand(0)
7.     if Op0. getOpcode() is Opcode2 then
8.       fNode←emitBuiltin1(Op,Op0)
9.         if hasOneUse(Op0) is true then
10.            costBefore←costBefore+cost(Op0)
11.      end if
12.      costAfter←cost(fNode)
13.      if costBefore>costAfter then
14.        fusionOp←fNode
15.      end if
16.    end if
17. break
18. OpcodeN )/ * 其他节点融合处理 * /
19. end case
```

算法 3 是中间表示阶段进行节点融合的算法伪代码,该算法将每一条 IR 视为一个节点并试图对节点进行节点融合优化。通过节点的 getOpcode 方法获取其操作码,将节点的操作码作为 case 的匹配值,从而达到对不同节点进行不同的节点融合处理。当节点的数据依赖关系满足匹配 case 的融合条件且融合后节点的代价 *costAfter* 小于节点融合前的代价 *costBefore* 时,通过发射内建函数调用的方式,完成节点融合优化,否则不进行节点融合优化。生成的内建函数调用 IR 将在 LLVM IR 的降级过程中被降级为目标平台相关的指令。

中间表示层的节点融合利用中间表示具有显式 use-def 链的特点,通过发射内建调用或节点操作 IR 的方式进行节点融合优化。但是由于中间表示应具有平台无关的属性,因此在该层进行平台相关的节点融合优化所生成的中间表示可能最终不会按照预期生成相应的机器指令。如果生成的融合节点不被目标平台后端所支持,则进行节点融合后的中间表示将影响后端的兼容性,而且也破坏了中间表示应具有的平台无关的特性。

4.2 DAG 合并阶段节点融合

从 LLVM IR 到 MIR 的过程,也是从平台无关的节点向平台相关的节点的降级过程。在这一降级过程中,LLVM IR 会转换成 DAG,利用图的匹配算法完成 LLVM IR 的降级处理。

DAG 降级阶段的节点融合优化是在 DAG 降级的过程中进行的节点融合处理,这些 DAG 合并进行了一系列的平台无关和平台相关的节点融合优化。完成这些节点融合优化,不仅可以简化 DAG,而且还可以提高目标平台代码的执行效率,减少运行时间,从而达到充分挖掘目标平台指令集潜力的目的。

由于 LLVM 良好的封装和高度抽象的特性,LLVM 提供了实现目标平台相关的节点融合优化的虚函数接口 PerformDAGCombine,LLVM 的目标平台后端只需继承 TargetLowering 类并实现 PerformDAGCombine 方法,就可以从通用的 DAG 合并流程,借助 TLI 信息进行自定义的节点融合优化,从而实现平台无关以及平台相关的节点融合。实现 DAG 阶段的节点融合优化的算法伪代码如算法 4 所示。

算法 4 DAG 阶段的节点融合

```
输入:(N,DCI) / * 待融合节点,DAG 信息 * /
输出:(fusionNode)/ * 融合节点 * /
1. DAG←DCI. DAG
2. costBefore←cost(N)
3. costAfter←0
4. fusionNode←N
5. case N. getOpcode() in
6. Opcode1 )
7.     Op0←N. getOperand(0)
```

```
8.    if Op0.getOpcode() is Opcode2 then
9.      fNode←DAG.getNode(Opcode3,N,Op0)
10.     if hasOneUse(Op0) is true then
11.       costBefore←costBefore+cost(Op0)
12.     end if
13.     costAfter←cost(fNode)
14.   if costBefore > costAfter then
15.     fusionNode←fNode
16.   end if
17. end if
18. break
19. OpcodeN ) /*其他 Opcode 处理*/
20. end case
```

算法 4 中 DAG 阶段的节点融合优化算法与算法 3 中的中间表示阶段的节点融合优化算法类似，但算法 3 进行平台无关的节点融合优化，而算法 4 既可以进行平台相关的节点融合优化（发射平台相关的节点），又可以进行平台无关的节点融合优化（发射平台无关的节点），这种差异是由其所处不同的编译阶段所引起的。算法 4 通过传入的待融合节点 N 的 Opcode 来区分不同的节点融合处理流程，若节点存在相应的 case 处理，则继续判断节点的数据依赖关系是否满足要求，然后计算融合前后的代价 $costBefore$ 和 $costAfter$，若融合后的代价 $costAfter$ 小于融合前的代价 $costBefore$ 则通过返回融合节点 $fNode$ 完成节点融合优化，否则返回原节点 N 不进行节点融合优化。

DAG 降级阶段的节点融合优化是在 DAG 降级的过程中进行的节点融合处理，它利用 DAG 有向无环的特点，根据拓扑排序算法对每一个节点进行处理。在这一降级过程中不仅提供平台相关的优化，而且也提供平台无关的优化。本阶段的节点融合优化是通过实现 LLVM 提供的虚函数接口，根据待融合节点的数据依赖关系来完成节点融合优化，该优化代码只在目标平台后端进行修改，既不影响 LLVM 的版本升级，也不影响中间表示的兼容性，而且还可以进行多次融合优化。

4.3　指令选择阶段节点融合

除了 4.2 节提到的在 DAG 合并阶段进行节点融合优化之外，还可以通过 Pattern[14] 匹配的方式进行节点的融合优化。虽然该过程也属于 DAG 降级阶段，但是不同于 DAG 合并阶段中的节点融合优化。Pattern 是 td 文件中的一个类，而 td 文件是 LLVM 用来定义后端指令集、寄存器和调用约定等的一套抽象描述。对应的 td 文件及其功能如表 1 所列。

表 1　后端架构相关的 td 文件

Table 1　td files related to the back-end architecture

文件名	描述
⟨target⟩.td	定义机器的特征
⟨target⟩InstrFormats.td	定义指令格式
⟨target⟩RegisterInfo.td	定义寄存器和寄存器类
⟨target⟩CallingConv.td	定义寄存器的调用约定
⟨target⟩InstrInfo.td	定义指令、模板和格式
⟨target⟩Schedule.td	定义指令流水

表 1 中的 td 文件描述了目标平台后端的特征信息，比如指令集、寄存器描述等。Pattern 类的定义如算法 5 所示。

算法 5　Pattern 的定义

```
1. class Pattern ⟨dag patternToMatch,list ⟨dag⟩ resultInstrs⟩ {
2. dag PatternToMatch=patternToMatch;
3. list ⟨dag⟩ ResultInstrs=resultInstrs;
4. list ⟨Predicate⟩ Predicates =[];
5. int AddedComplexity=0;
6. }
7. class Pat⟨dag pattern,dag result⟩:Pattern⟨pattern,[result]⟩;
```

从算法 5 中可以看到，指令 Pattern 的两个输入 patternToMatch 和 resultInstrs 都是 dag 类型的变量，其中 PatternToMatch 是要匹配的图（即模板输入），而 ResultInstrs 定义了从 PatternToMatch 转换成的结果图（即模板输出）。通过指令 Pattern 可以完成节点的融合优化，该过程是在指令选择阶段完成的。以浮点乘加指令的 Pattern 举例说明如算法 6 所示。

算法 6　浮点乘加指令的 Pattern 定义

```
1. def :Pat⟨(fadd (fmul F8RC:$A,F8RC:$B),F8RC:$C),(FMA
   F8RC:$A,F8RC:$B,F8RC:$C)⟩;
2. def :Pat⟨(fadd F8RC:$C ,(fmul F8RC:$A,F8RC:$B)),
   (FMA F8RC:$A,F8RC:$B,F8RC:$C)⟩;
3. def :Pat⟨(fneg (fsub (fneg F8RC:$C),(fmul F8RC:$A,F8RC:
   $B))),(FMA F8RC:$A,F8RC:$B,F8RC:$C)⟩;
4. def :Pat⟨(fneg (fsub (fneg (fmul F8RC:$A,F8RC:$B)),F8RC:
   $C)),(FMA F8RC:$A,F8RC:$B,F8RC:$C)⟩。
```

算法 6 中 fadd 是浮点加节点，fmul 是浮点乘节点，FMA 是浮点乘加节点，fneg 是浮点取负节点，fsub 是浮点减节点。$A，$B，$C 分别表示变量 A，B，C。以上 4 个 Pattern 分别对应的表达式为 $A\times B+C$，$C+A\times B$，$-(-C-A\times B)$，$-(-A\times B-C)$，其中 A，B，C 都为浮点类型的变量，F8RC 表示双精度浮点类型寄存器。这 4 个表达式都可以化简为 $A\times B+C$ 的形式，所以都可以通过浮点乘加指令来完成。Pattern 匹配通过在⟨Target⟩InstrInfo.td 中编写相应的 Pattern 模板来完成节点融合，Pattern 的匹配过程如图 6 所示。

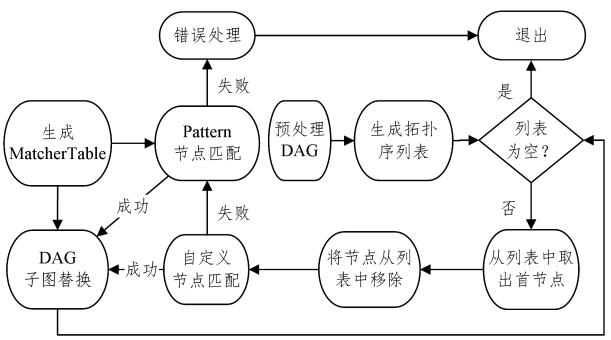

图 6　Pattern 匹配流程

Fig. 6　Matching process of Pattern

图 6 中 Pattern 的匹配流程为先取得待匹配节点，判断节点偏移表是否为空，如果为空则根据 Pattern 匹配表（MatcherTable）信息构建偏移表，否则从偏移表中取出节点的偏移值；如果偏移值为零则进行错误处理，将显示无法选择节点错误，然后退出编译程序；否则检查节点类型及约束，检查以该节点为根的 Pattern 子图能否和 DAG 相匹配，若成功则进行 DAG 子图替换将相匹配的子图（PatternToMatch）替换为目标子图（ResultInstrs），否则从 MatcherTable 中获取以该节点为根节点的下一 Pattern 子图的偏移，然后判断偏移是

否为零，进行循环处理。

Pattern 匹配方式的节点融合优化是利用 Tablegen 工具将 td 格式的 Pattern 代码生成 MatcherTable，然后利用通用的 Pattern 匹配流程完成节点融合优化。该种方式只需添加 Pattern 代码到目标平台相关的指令 td 文件中，不会影响 LLVM 的版本升级和其他后端，但是该种方式由于无法有效获取子图节点的数据流信息，因此不能进行节点融合前后的代价评估，只能尽最大努力进行节点融合优化，因此可能会生成质量较差的目标平台代码。

5 实验

本文采用国产平台申威处理器 1621 为实验平台，以 CLANG[1] 和 FLANG[15] 为前端编译器，LLVM 为后端编译器，编译器的版本为 7.0，所使用的测试集为 SPEC CPU2006。在该种配置下，进行两个实验：不进行节点融合优化和进行节点融合优化的实验。通过这两个实验的结果对比，评估节点融合的优化效果。SPEC CPU2006 测试集包含 13 道定点程序，18 道浮点程序，由 C，C++以及 Fortran 编程语言编写。本实验对其中 29 道程序进行测试，编译优化选择-O3 -static，train 规模，单进程运行，每道题测试 3 次，然后取平均运行时间。测试结果如表 2 所列。

表 2 节点融合实验结果

Table 2 Experiment result of node fusion

程序名称	编程语言	节点融合优化前/s	节点融合优化后/s	加速比
400.perlbench	C	114.17	103.09	1.11
401.bzip2	C	164.99	142.76	1.16
403.gcc	C	3.98	3.67	1.09
410.bwaves	F77	129.04	114.83	1.12
416.games	Fortran	426.97	340.07	1.26
429.mcf	C	83.84	83.41	1.01
433.milc	C	65.09	64.74	1.01
434.zeusmp	F77	118.55	105.90	1.12
435.gromacs	C、Fortran	522.21	327.91	1.59
436.cactusADM	F90、C	44.85	32.07	1.40
437.leslie3d	F90	268.94	238.38	1.13
444.namd	C++	38.3	31.76	1.21
445.gobmk	C	417.18	371.47	1.12
447.dealII	C++	156.50	143.77	1.09
450.soplex	C++	23.77	20.30	1.17
453.povray	C++	29.46	26.57	1.11
454.calculix	F90、C	3.75	3.24	1.16
456.hmmer	C	170.90	168.91	1.01
458.sjeng	C	535.37	489.92	1.09
459.GemsFDTD	F90	195.26	173.28	1.13
462.libquantum	C	6.67	5.79	1.51
464.h264ref	C	274.19	232.26	1.18
465.tonto	F95	1678.97	1567.78	1.07
470.lbm	C	191.30	144.46	1.32
471.omnetpp	C++	278.21	267.13	1.04
473.astar	C++	315.63	283.98	1.11
481.wrf	F90、C	432.90	547.45	0.79
482.sphinx3	C	31.73	31.23	1.02
483.xalancbmk	C++	570.09	548.46	1.04
平均加速比				1.13

从表 2 可看出，节点融合优化对大部分程序具有明显的加速效果，加速比最高的程序有 1.59 倍的加速，平均加速比为 1.13。通过对加速效果最好的程序 435.gromacs 进行性能分析发现，程序执行过程的时间消耗有 70%集中在函数 inl1130 上，进一步分析函数 inl1130 的代码发现其中包含多个开方函数和浮点乘加表达式。通过节点融合优化可以对开方函数和浮点乘加表达式进行节点融合优化，使其转换为针对国产申威平台的扩展指令，从而减少程序的执行时间，提高程序运行效率。程序能够加速的主要原因是通过节点融合减少了冗余指令和用扩展指令来替代原有的指令提升了程序的执行效率，使系统的资源得到了更加充分的利用。

虽然多数程序都具有正加速的效果，但是其中也有个别程序的加速比小于 1.00，比如程序 481.wrf 的加速比为0.79。程序未进行节点融合优化前其运行(进程从开始执行直到结束的时间)时间为 432.90 s，进行节点融合优化后其运行时间增加到 547.45 s，节点融合优化显著地增加了程序的运行时间。通过对程序进一步分析发现，程序执行的用户时间(进程花费在用户模式中的 CPU 时间)在进行节点融合前后差异不大，分别为 354.23 s 和 331.82 s。从用户时间的减少可以看出，节点融合优化具有加速效果。程序 481.wrf 程序下降的主要原因是进行节点融合优化后，该程序的系统时间(进程花费在内核模式中的 CPU 时间，代表在内核中执行系统调用所花费的时间)显著增加，由未进行节点融合前的 78.21 s 增加到 215.63 s，是未进行节点融合优化所花系统时间的 3 倍。系统时间的大幅度增加导致了 481.wrf 程序进行节点融合优化后性能下降，说明其中存在不合理的节点融合优化。

结束语 在中间表示阶段、DAG 合并阶段和指令选择阶段进行节点融合优化各有优缺点：在中间表示阶段进行融合，虽然有比较丰富的中间表示信息，但是无法获知后端是否支持该种优化，所以存在融合节点被拆分为多个节点的风险；DAG 合并阶段的节点融合，该阶段既有丰富的中间表示信息，又有后端的指令集信息，既可以进行平台无关的优化，又可以进行平台相关的融合优化，还能进行多次节点融合优化；指令选择阶段的 Pattern 匹配节点融合，该阶段通过通用的 Pattern 匹配流程进行 DAG 子图的融合优化，由于无法获取 DAG 子图节点之间完整的数据依赖关系，导致无法进行融合前后的代价评估，只能尽最大努力进行节点融合，因此容易进行过度优化，反而不利于性能的提升。

本文实现并介绍了 LLVM 在中间表示阶段、DAG 合并阶段和指令选择阶段进行的节点融合优化，并通过实验说明了节点融合所带来的性能提升，结果表明节点融合优化有利于提高编译器性能，减少程序运行时间，优化后最大加速比为 1.59，平均加速比为 1.13。下一步的工作主要是分析 SPEC CPU2006 部分程序倒加速的原因，完善 LLVM 编译器的节点融合优化。

参考文献

[1] LATTNER C. LLVM and Clang: Next generation compiler technology[C]// The BSD Conference. 2008, 5.

(下转第 595 页)

logy and trust in the sharing economy [J]. Electronic commerce research and applications,2018,29(1):50-63.
[13] ZHOU J,LI W Y,GUO G. Patent Situation Analysis of Blockchain Technology[J]. Telecommunications Network Technology,2017(3):37-42.
[14] WANG C L,WANG Y D,QIN Q,et al. Supply chain logistics information ecosystem model based on blockchain [J]. Information Studies:Theory & Application,2017(7):115-121.
[15] CHEN L,SUN W,LI H L. Research on Financial Risk Prevention of Supply Chain Based on Blockchain Technology[J]. Hebei Enterprise,2018(2):32-33.
[16] ZHANG J L,LUN Z W. Research on the Combination of Blockchain Technology and Supply Chain Finance[J]. Co-Operative Economy & Science,2017(21):58-59.
[17] WU J. The Application of Blockchain Technology in Supply Chain Finance-Based on the Perspective of Information Asymmetry[J]. Logistics Technology,2017,36(11):121-124.
[18] ZHU X X,HE Q S,GUO S Q. Application of Blockchain Technology in Supply Chain Finance [J]. China Business and Market,2018,32(3):111-119.
[19] ZHAO M H,ZHANG L,QI J. Blockchain-based social Internet of things trusted service management framework [J]. Telecommunications Science,2017,33(10):19-25.

KE Yu-jing, born in 1999, bachelor. Her main research interests include network and information security, and block chain technology.

JING Mao-hua, born in 1977, Ph.D, lecturer. Her main research interests include automata principle, network and information security.

(上接第 566 页)

[2] LATTNER C,ADVE V. LLVM:A compilation framework for lifelong program analysis & transformation[C]// Proceedings of the International Symposium on Code Generation and Optimization:Feedback-directed and Runtime Optimization. IEEE Computer Society,2004:75.
[3] ZHAO J,NAGARAKATTE S,MARTIN M M K,et al. Formalizing the LLVM intermediate representation for verified program transformations[C]// Acm Sigplan Notices. ACM,2012,47(1):427-440.
[4] PANDEY M,SARDA S. LLVM cookbook [M]. Packt Publishing Ltd.,2015.
[5] FRASER C W. A compact,machine-independent peephole optimizer[C]// Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM,1979:1-6.
[6] SPRADLING C D. SPEC CPU2006 benchmark tools[J]. ACM SIGARCH Computer Architecture News,2007,35(1):130-134.
[7] GUOBIN Y E. Getting to know the LLVM compiler[D]. Master's thesis,The University of Edinburgh,2011.
[8] LATTNER C. Introduction to the llvm compiler infrastructure [C]// Itanium Conference and Expo. 2006.
[9] PANDEY M,SARDA S. LLVM cookbook [M]. Packt Publishing Ltd,2015.
[10] CYTRON R,FERRANTE J,ROSEN B K,et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages and Systems (TOPLAS),1991,13(4):451-490.
[11] CALLAHAN D,COOPER K D,KENNEDY K,et al. Interprocedural constant propagation[C]// ACM SIGPLAN Notices. ACM,1986,21(7):152-161.
[12] COCKE J. Global common subexpression elimination[J]. ACM Sigplan Notices,1970,5(7):20-24.
[13] LATTNER C,ADVE V. The LLVM instruction set and compilation strategy[J]. CS Dept.,Univ. of Illinois at Urbana-Champaign,Tech. Report UIUCDCS,2002.
[14] HOKENEK E,MONTOYE R K,COOK P W. Second-generation RISC floating point with multiply-add fused[J]. IEEE Journal of Solid-State Circuits,1990,25(5):1207-1213.
[15] OSMIALOWSKI P. How The Flang Frontend Works:Introduction to the interior of the Open-Source Fortran frontend for LLVM[C]// Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC. ACM,2017:1.

HU Hao, born in 1994, postgraduate. His main research interests include information security.