毕昇编译器的创新与实践

高耀清
华为编译器与编程语言实验室
yaoqing.gao@huawei.com

华保健
中国科学技术大学软件学院
bjhua@ustc.edu.cn

摘要—计算产业的场景诉求、硬件架构和微架构的不断改进和创新，如软硬件协同/软件定义硬件、**ISA** 领域扩展等新的研究趋势和进展，不断对编译器设计和实现提出新的研究挑战。为应对这些研究挑战，毕昇编译器通过结合业务诉求及工业界、学术界研究热点和进展，在 **LLVM** 编译器的先进架构的基础上，通过在编译器前端、中端、后端引入技术创新，针对性的构建关键差异化竞争力，快速迭代演进，实现编译器使能鲲鹏和昇腾极致性能，支持典型场景性能优势和技术生态构建。同时面向未来，毕昇编译器积极拓展并布局新的创新研究方向。本文在总结分析编译器技术演进的基础上，系统介绍华为毕昇编译器的技术创新与实践，分析核心关键技术特性，并指出进一步演进和创新的技术方向。

关键词—毕昇编译器；异构计算；超级优化
中图法分类号—G312

I. 引言

编译器是一种重要的基础软件，它将面向编程人员的高级语言翻译成硬件能够执行的低级机器语言。在翻译的过程中，编译器对程序进行复杂的分析及优化，改进程序的执行性能。同时，编译器还需要兼顾高层编程语言实现的正确性和处理底层目标体系结构的复杂性。操作系统、数据库、网络协议栈、人工智能及机器学习框架等高层软件，都需要底层编译器进行有效支撑，由于编译器在整个软件栈中的重要作用，它被称为软件开发中的皇冠。

编译器也是计算机科学中理论和实践结合最紧密的学科，围绕编译器的理论研究和工程实践，理论界和工业界已经研究了丰富的编译器设计理论并进行了大量卓有成效的工程实践，这些理论和实践包括程序设计语言类型、文法 [1][2][3]、类型系统、基于格和不动点的数据流分析 [4][5][6]、图论及应用 [7][8]，约束求解等。由于编译器设计中的许多问题，例如后端的寄存器分配 [9][10] 或指令调度 [11][12] 等，都是 NP 难问题，对这些问题的探索仍是编译器研究的热点问题。

编译器工程也已经取得了令人瞩目的成果和进展，除了开源社区中应用非常广泛的开源编译器外，很多大学、研究机构和公司等也都维护或发布了面向不同语言和不同目标体系结构的开源或商业版本的编译器，这些编译器工程实践对于推进学术研究或支撑商业成功都起到了非常关键的作用。

随着应用新场景诉求的变化，硬件架构的演进和基础理论及技术的发展，编译技术的理论研究创新和实践探索都取得了非常大的进展。这些新的编译器设计和实现创新和进展有效支撑了大数据、云计算、异构计算、深度学习等新的应用场景。但是，目前尚没有最新文献，对编译器设计的新成果、新进展进行系统性介绍和总结，以及对新方向进行系统性展望。

针对这一问题和挑战，本文的主要目标是基于华为毕昇编译器 [13]，对编译器的技术创新和实践进行系统总结和介绍，并探讨未来发展方向。本文主要内容包括：1）首介绍编译器的发展史，编译器架构、编译器业界现状等内容，并通过深入分析当前最流行的 LLVM 开源编译器的社区最新进展等，对编译器重要概念进行总结。2）重点讨论华为毕昇编译器的架构，尤其是其使用的先进编译优化技术如循环优化、自动向量化等，并讨论了基于 AI 的编译器自动调优等内容；这些特性对支持毕昇编译器的成功起到了关键作用。3）通过对业界趋势的分析，探讨了华为毕昇编译器未来规划可能发展的几个重要方向、以及将带来的重要价值。

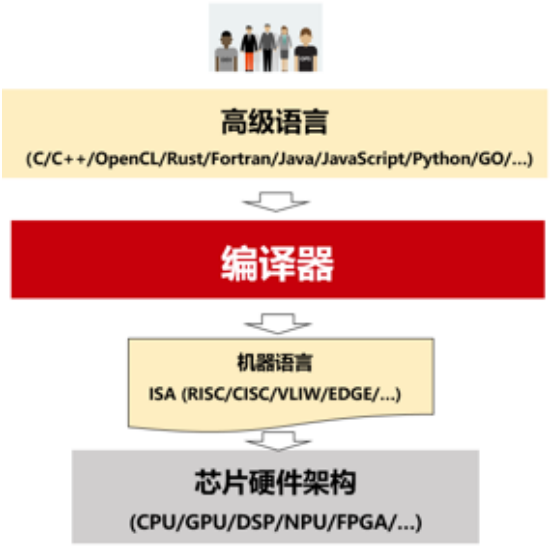


图 1: 编译器场景

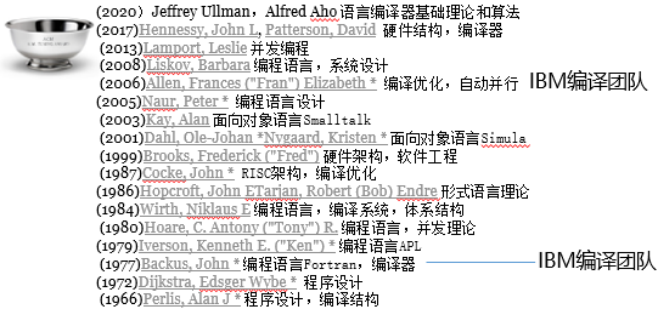


图 2: 图灵奖中编译器相关研究获奖情况

II. 编译器简介

A. 编译器基本概念

编译器（compiler）是一种计算机程序，它将一种编程语言（源语言）写成的源代码转换成另一种等价的编程语言（目标语言）编写的程序。它的主要目的是将用便于人编写、阅读、维护的高级编程语言所写的源程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。源语言一般为高级语言（High-level language），如 C、C++、Java、Rust、Go 等，而目标语言则是汇编语言或目标机器的目标代码（Object code），有时也称为机器代码（Machine code），如 x86、ARM、RISC-V 等 [14]。

编译器结构可分解为两个主要部分：前端（Front-end）和后端（Back-end）。前端和后端有明确的分工：前端专注于理解源语言程序，把源程序分解为多个词法单元，并检查这些词法单元是否满足语言规定的语法结构，检查该源程序是否符合程序语义的规定，生成一种中间表示（Intermediate Representation，IR）。

后端专注于将中间表示映射到目标机器上，其要完成的任务包括为源程序的语法结构选择合适的机器指令，以及为源语言的变量分配适当的目标机器的资源，等等。

现代的优化编译器一般还会引入中端（Middle-end），它是一个和具体语言以及具体目标机器相对无关的中间阶段。引入中端的主要作用和优势有两个：第一，引入中端可以使得编译器的前端和后端解耦，即多个语言的前端都可以生成公共的中端代码，而中端代码进一步又可以生成不同目标机器的代码；第二，在中端上，可设计并实现与语言以及目标机器无关的程序优化算法，对程序的性能、规模或其它指标进行通用优化。

基于编译器复杂性和应用场景的广泛性，编译器的设计要同时满足以下目标：向上面向应用开发者编程产能，向下支持多样化硬件极致性能，解决 3P 的问题，具体包括：

1) 开发者编程产能（Productivity）：屏蔽硬件架构信息，易于程序员高效编程；
2) 使能硬件极致性能（Performance）：实施静态动态编译优化，使能硬件极致性能，并有利于软硬协同设计；
3) 软件兼容可移植性（Portability）：支持开放式场景友商生态迁移，解决硬件跨代源码/二进制级兼容以及编译器升级软件行为和性能兼容。

同时，编译器还需要安全可靠（Safety and Reliability），能够通过动静态程序分析做到检错纠错，保证程序代码的安全可靠。

由于编译器理论和实践的重要性，编译器一直在计算机科学中处于核心地位。以计算机科学的最高奖—图灵奖为例：第一个图灵奖就颁发给了编程语言和编译器专家 Alan Perlis 教授；此后，平均约每三到五年就有编程语言编译器相关领域的研究获奖。

B. 编译器发展史

自 20 世纪 50 年代中期以来，编译器设计就一直是计算机科学重要研究领域。Fortran 编译器是第一个被广泛使用的高级语言编译器，它由 IBM 开发，是一个多遍系统，其设计和实现引入了现代编译器中的很多

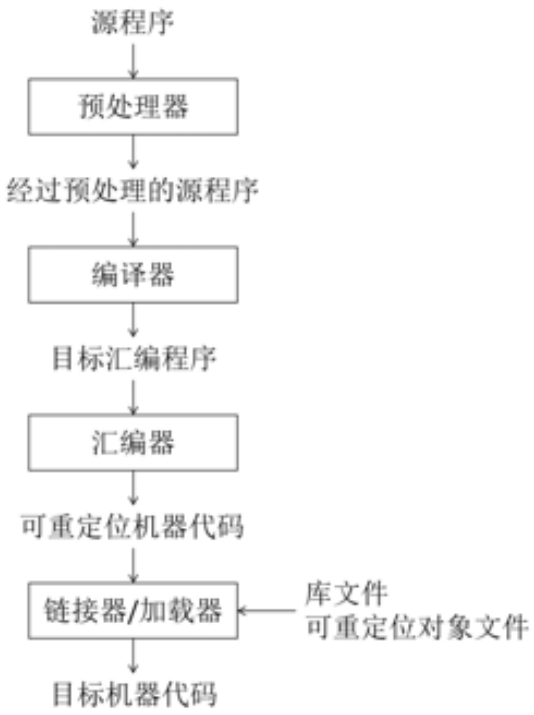


图 3: 编译器典型架构

重要概念，如独立的词法分析器、语法分析器，以及包括寄存器分配在内的很多程序优化算法等。

在 20 世纪六七十年代，研究者研究并构建了许多有影响力的编译器。这其中包括经典的优化编译器 FORTRANH[15][16]、Bliss-11 和 Bliss-32 编译器 [17]，以及可移植的 BCPL 编译器 [18]，等等。这些编译器可以为各种复杂指令集计算机（Complex Instruc-tion Set Computer，CISC）体系结构生成高质量的目标代码。

在 20 世纪 80 年代，精简指令集计算机（Reduced Instruction Set Computer，RISC）体系结构的问世对编译器设计产生了深远影响。该体系结构要求编译器设计者不仅需要跟踪新的程序设计语言特征，还要研究并设计新的编译算法，以便能最大限度地发挥新硬件的计算能力。这些趋势导致了新一代编译器 [19][20][21] 更加专注于强有力的中端代码优化技术、后端代码优化以及代码生成技术的研究。这一阶段编译器的典型架构如下图所示，现代 RISC 体系结构上的编译器仍然遵循该架构模型。

从 20 世纪 90 年代开始，编译器研究人员开始专注于处理微处理器系统结构中提出的挑战，包括处理器中多功能部件、内存延迟和代码并行化，等等。事实证明，

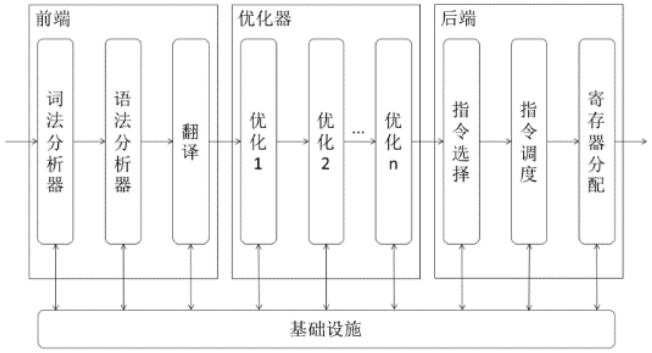


图 4: 优化编译器典型架构

20 世纪 80 年代提出的针对 RISC 架构的编译器的结构和组织，仍然具有足够的灵活性来应对这些挑战，因此，研究人员可以通过构建新的遍（Pass），插入到编译器的优化器和代码生成器中，来应对体系结构提出的挑战 [22]。单靠各组织机构独立开发全自研封闭编译器成本越来越高昂，这个时期，理查德 · 马修 · 斯托曼决定开发对未来影响深远的 GCC 开源编译器。作为 GNU 系统的官方编译器（包括 GNU/Linux 家族），它也是编译与创建其他操作系统的主要编译器，包括 BSD 家族、Mac OS X、NeXTSTEP 与 BeOS 等。采用三段式架构的 GCC 是跨平台软件的编译器首选。有别于局限于特定系统与运行环境的编译器，GCC 在所有平台上都使用同一个前端处理程序，产生一样的中间表示，在各个其他平台上编译，并正确无误的输出程序 [34]。

C. LLVM 开源编译器介

由于体系架构，编程语言种类等不断增加，GCC 在快速适配新的需求，构建竞争力上也开始显得不够灵活。LLVM 开源编译器在编译器三段式架构基础上，采用模块化设计和库实现，其生态兼容、友好的软件许可证，社区活跃度、学习成本等因素被学术和工业界追捧，已经具备明显的主导趋势。

LLVM 是一种涵盖多种编程语言和目标处理器的编译器，以 C++ 编写而成，包括了前端、后端、优化器、众多的库函数以及若干的模块，对开发者保持开放，并兼容已有脚本。LLVM 已作为实现各种静态和运行时编译语言的通用基础结构。目前有 140 多个公司参与 LLVM 项目，涵盖不同产业和硬件架构，近 10 年 LLVM 社区持续保持高热度。

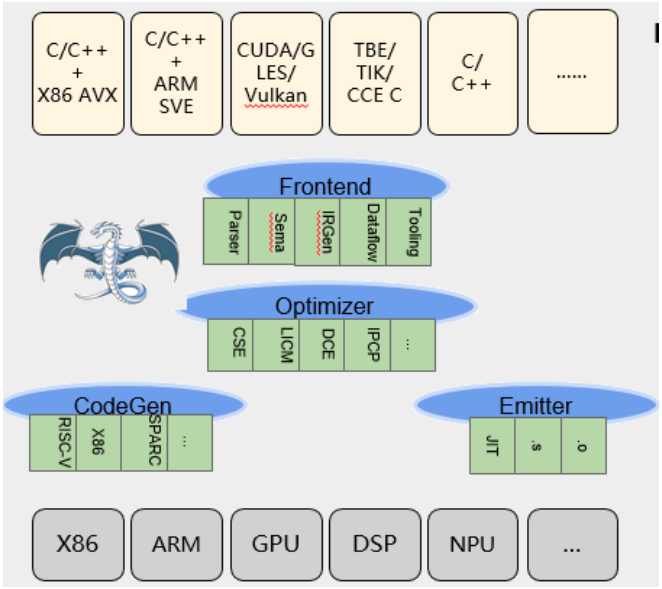


图 5: LLVM 架构图

公司	编译器	应用场景
apple	Swift/Objective-C	终端
Google	Android 编译器工具链	终端
高通	Android 编译器 + Hexagon编译器	终端+5G
IBM	XL C/C++编译器	高性能计算/通用服务器
AMD	AOCC Fortran/C/C++编译器	高性能计算/通用服务器
ARM	AHC编译器/Mali GPU编译器	终端/高性能计算/通用服务器
Intel	DPC++编译器	高性能计算/通用服务器
其它	Facebook，微软，英伟达，Xilinx，Synopsys... ...	

图 6: 毕昇编译器架构简图

其中主要贡献者主要来自于 Apple、Google 和芯片大厂，用于支持多种标准语言和自研语言如 Swift 、Rust、GO、及使能不同架构的芯片如 X86、ARM、RISC-V 等性能，同时也衍生出一些创新的技术如：MLIR 等 [25] [26]。

目前行业大厂纷纷加入 LLVM 编译器的布局，除了参与社区贡献外，基于 LLVM 架构，通过差异化竞争力的编译技术，构建自己品牌的闭源编译器：Apple、Google、高通、IBM、AMD、ARM、Intel、Facebook、微软，华为等都对 LLVM 进行支持；涉及的场景包括终端、5G、计算/HPC/服务器等广泛领域。

III. 华为毕昇编译器

A. 毕昇编译器介绍

毕昇编译器是华为公司推出的高性能多样化算力编译器，它基于开源的 LLVM 编译框架 [27]，针对应用

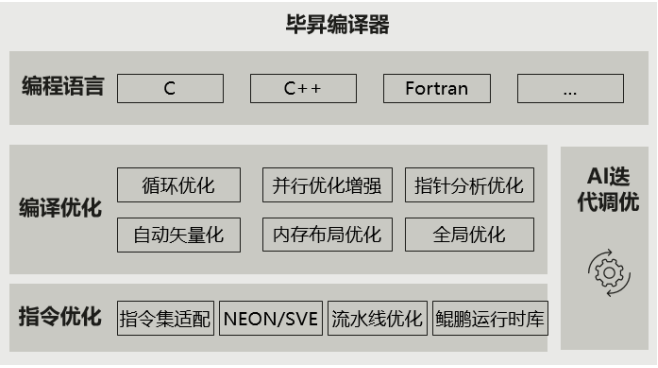


图 7: 毕昇编译器架构简图

场景，编程语言和硬件架构，构建差异化竞争力的通用编译优化技术，架构相关和应用场景相关的深度协同优化等。图 7 简要展示了毕昇编译器的总体架构。不同编程语言编写程序作为编译器的输入，被转换为内部统一的中间表示，编译器组织了一系列优化遍，如循环优化、内存布局优化和自动矢量化等，对中间表示进行优化变换，挖掘程序中的并行性和局部性，充分利用内建加速指令等硬件特性。同时毕昇编译器还集成了 Autotuner 特性，对静态优化过程中无法确定的优化参数进行自动调优。最终，编译器输出针对鲲鹏平台的高效可执行程序。

B. 毕昇编译器关键竞争力技术

毕昇编译器在高性能编译算法、定制加速指令集优化、AI 迭代调优 (Autotuner) 多个方向重点布局关键竞争力。毕昇编译器使用先进的高性能编译算法，挖掘程序中的并行性和局部性；使用架构相关指令优化，选择定制的指令集加速程序；对于静态优化时无法确定的调优选择，使用 AI 迭代调优自动地进行优化。

1) 高性能编译优化算法: 传统芯片追求通用性，微结构复杂，导致软件难以发挥出硬件性能。为了挖掘晶体管潜力，针对特定场景，在性能功耗优先的前提下，按领域特征针对性优化，通过软件的复杂度换取性能功耗比提升，以此来实现进一步的性能收益是业界目前主要的手段。

2017 年，计算机架构领域两位重量级人物 David Patterson 与 John Hennessy 在斯坦福大学发表演讲时举的例子，通过使用多种架构设计及编译优化算法技术实现了程序性能的大幅提升 [28]，图 8 展示了矩阵乘计算在不同实现和不同优化技术下的性能加速比。

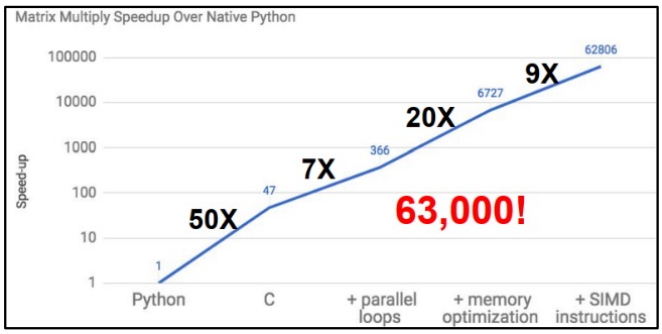


图 8: 毕昇编译器架构简图

为了获取更好的性能，毕昇编译器同样从多个方向发力，实现应用对硬件的充分利用，目前重点发力的方向有：

1) 针对特定领域、不同层次更有效的并行，包括：
   - 循环并行；
   - 指令集并行 ILP、内存层次并行 MLP；
   - 超标量、多线程、多核、SIMD、STMD；
   - VLIW vs 投机，乱序；
2) 空间/时间局部性下更有效内存利用；
   - 用户控制 vs 高速缓存；
   - cache 访问、内存搬移、函数分块等；
3) 消除不必要数据精度；
   - 低精度 FP 取代 IEEE；
   - 32 64 位整数到 8 16 位整数；
   - 混合精度计算模式；

基于以上思路，毕昇针对性增强编译优化算法。

3.2.1.1 循环并行优化

循环内的语句通常被多次执行，针对循环进行优化能够得到显著的收益。因此循环优化是编译器重要的性能提升手段，具有广泛和多样化的优化算法。如《计算机体系结构将迎来一个新的黄金时代》[28] 中提到，通过循环并发，用例获得了大幅度的性能提升。编译器可以通过不同的算法来提升循环的性能，如：提高缓存利用率、降低寄存器压力、减少动态指令数和复用不同迭代加载或计算值暴露其它优化的机会：向量化、指令调度等。毕昇采用了多种循环优化手段，其中重要的算法如下：

1：Loop Unroll and jam

该优化技术通过展开嵌套循环的外层循环，融合内层循环的循环体 (body)，提升 cache 的利用率，如下例所示。

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    a[j] += b[i][j] + b[i-1][j];
  }
}
```

编译优化后：

```
for (int i = 0; i < n-n%2; i+= 2) {
  for (int j = 0; i < m; j++) {
    a[j] += b[i][j] + b[i-1][j];
    a[j] += b[i+1][j] + b[i][j];
  }
}
```

通过以上转换，Cache 命中率能得到有效改善。

2. Loop Fusion/Distribution

Loop Fusion 将两个循环的循环体 (body) 合并，寻找创建一个循环的机会。这样的优化好处在于，通过合并寻找可重用值、暴露指令调度机会；Loop Distribution 是 Loop Fusion 的逆过程，将一个循环的循环体 (body) 划分为两个单独的循环体，这样优化的好处在于暴露矢量化优化的机会，同时找到其它优化空间。下面的例子通过优化将前两个语句保存在一起，以重用加载的值，同时分离出不可向量化的第三个语句。

```
for (int i = m; i < n; i++) {
  a[i] = b[i-1] + b[i];
  c[i] = b[i-1] - b[i];
  d[i-1] = d[i-2] + 1;
}
```

编译优化后：

```
int ub = n - n%2;
for (int i = 0; i < ub; i += 2) {
  a[i] = b[i-1] + b[i];
  a[i+1] = b[i] + b[i+1];
}
if (i < n)
  a[i] = b[i-1] + b[i];
```

3. Loop Unrolling

循环展开技术通过复制循环的循环体 (body)，减少循环迭代次数，从而减少分支跳转和循环边界检查。

另外，展开后的循环体内可能还会新增可重用值，暴露更多指令调度机会。

```
for (int i = 0; i < n; i++) {
  a[i] = b[i-1] + b[i];
}
```

编译优化后：

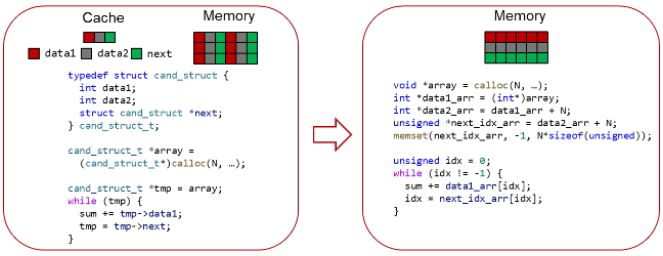
```
for (int i = m; i < n; i++) {
  a[i] = b[i-1] + b[i];
  c[i] = b[i-1] - b[i];
}

for (int i = m; i < n; i++) {
  d[i-1] = d[i-2] + 1;
}
```

4. Loop unswitch

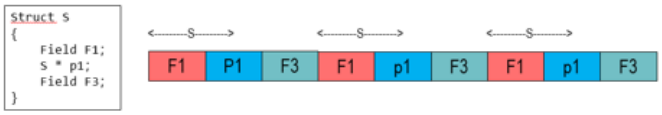
Loop unswitch 通过外提循环内判断条件不发生变化的条件分支语句，减少分支跳转，提供更多局部优化的时机。

3.2.1.2 内存优化

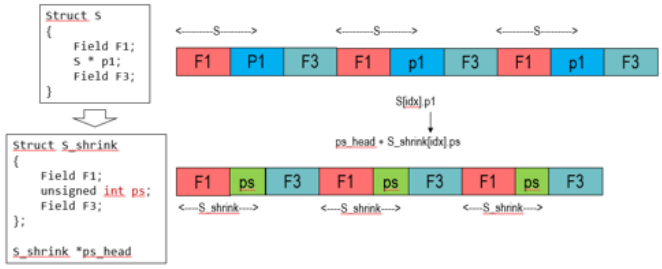
1：结构体内存布局优化。毕昇编译器针对结构体内存优化布局做了强化，结构体内存布局优化基于全程序（Whole-program）优化，用以提高缓存（cache）利用率。优化的主要手段是将结构体数组转换为数组结构体。结构体可以是显式的，也可以通过检查循环中的数组使用情况来推断它们。



2：结构体指针压缩优化。结构体指针压缩优化技术通过降低内存使用空间，提升 D-cache 的命中率。下文以一个简单的结构体优化示例来具体解释该优化技术。如下图所示，指针成员 P1 占据 8 字节

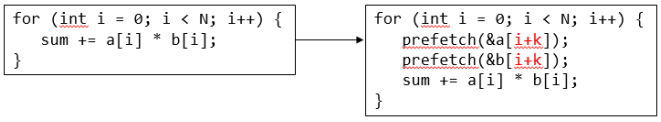


通过将域成员指针压缩，指针外提，减少了每个结构体 node 的内存体积，从而提升 cache line 中可以存储的结构体数据，进而提升 cache 命中率



通过全局结构体指针 ps_head 和 ps 变换为相对基址的偏移组合访问原来的域指针成员。

3：预取优化。软件预取是一种通过插入预取指令提前从内存中读取所需数据的优化技术。插入预取效果取决于“提前时间”，数据太早到达会浪费宝贵的 cache 空间，而数据太晚到达则仍需要等待访存过程。因此预取的效果取决于预取的“提前量“。在选择软件预取“提前量”的过程中需要综合考虑 Cache line 大小、访存延时、循环大小等。毕昇编译器针对鲲鹏微架构特征调整软件预取参数，选择精确的预取时机，从而提升 D-cache 的命中率。



3.2.1.3 浮点精度调优

HPC 应用（如气象 WRF，Grapes 等）迁移至鲲鹏服务器后，往往存在计算结果与原有数据结果比对不一致的场景。如下图所示，计算结果的精度差异主要来自四个方面：数学库、编译器优化、架构指令差异、运行环境。

针对四种可能导致精度差异的主要因素，对应的解决思路如下：

- 编译器优化差异：1. 支持 fp-model 等精度控制选项 2. 编译器优化行为调整及精度优化；
- 指令架构差异：结合转码工具分析指令差异，使用软浮点模拟等手段消除差异；

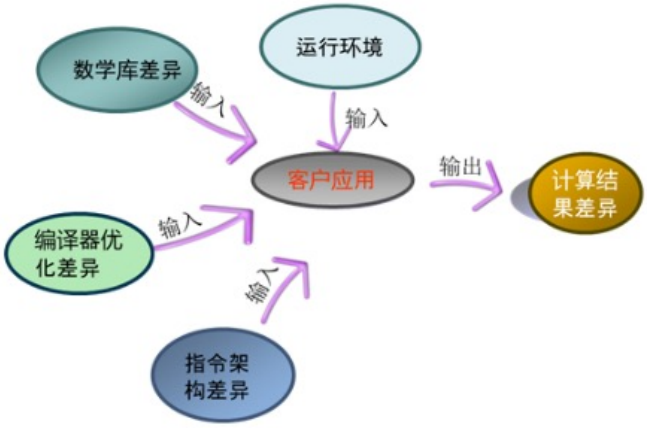


图 9: 精度差异原因

- 数学库差异：改写数学库函数，或使用不同精度数学库；
- 运行环境差异：集群和多核环境差异（如 MPI 及 OpenMP 等）；
- 优化及改进不同精度模式下的浮点编译算法。

场景一：乘法指令替换除法，可以使能流水线化；调节牛顿迭代，按需调整结果精度。

场景描述（单精度）：

```
fdiv   s8, s8, s10

(~ 11cycles)
```

预期优化结果（default）：两次牛顿迭代

```
frecpe  s2, s8
frecps  s4, s2, s8
fmul    s2, s2, s4
frecps  s4, s2, s8
fmul    s2, s2, s4
fmul    s10, s2

(pipeline with II=6)
```

预期优化结果（sdiv=1）：降低精度，一次牛顿迭代

```
frecpe  s2, s8
frecps  s4, s2, s8
fmul    s2, s2, s4
fmul    s10, s2

(pipeline with II=4)
```

场景二：乘加指令合并，通过控制（复数）乘加指令生成数量，进行精度优化。乘加指令合并：

场景描述（a*b+c）：

选项：-ffp-contract=on
预期优化结果：

```
fmul    s1, s1, s2
fadd    s0, s0, s1
```

→

```
fmadd s0, s1, s2, s0
```

矢量化 预期优化结果：

```
fmul    s1, s1, s2
fadd    s0, s0, s1
```

→

```
fmla    v4.4s, v2.4s, v1.4s
fmla    v5.4s, v3.4s, v1.4s
```

复数乘加指令合并：

场景描述 g = (a+bi)*(c+di)；
其中e = a + bi, f = c+ di

使用fcmla指令：
g = fcmla (e, f, 0),
g = fcmla (e, f, 90)

```
fmul    v5.2d, v0.2d, v1.2d
fmul    v6.2d, v0.2d, v2.2d
fmls    v5.2d, v2.2d, v18.2d
fmla    v6.2d, v1.2d, v18.2d
```

↓

```
movi    v1.2d, #0000000000000000
fcmla   v1.2d, v0.2d, v3.2d, #0
fcmla   v1.2d, v0.2d, v3.2d, #90
```

2) 加速指令集优化： 定制加速指令是自研芯片一个重要的竞争力构建手段，编译器在指令生成和指令选择的优化过程中发挥了重要作用，本小节介绍其中向量化定优化场景。

在并行计算中，自动矢量化是自动并行化的一个特殊场景，其中计算机程序从一次处理一对操作数的标量实现转换为一次操作中同时处理多对操作数的矢量实现。例如，基于 AArch64 的鲲鹏 920 处理器，具有 32 个 128 bits 的矢量寄存器，可以一次操作 4 路 32 位或者 2 路 64 位的数据。毕昇编译器重点优化了循环矢量化及 SLP 矢量化，充分保持程序局部性，高效提升计算密集型场景的性能。

下图中展示了循环向量化原理。

```
void foo(int *a, int *b,
    int *c, int n) {
  for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
}
```

→

```
void foo(int *a, int *b,
    int *c, int n) {
  for (i=0; i<n/4; i+=4) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
  }
}
```

→

```
void foo(int *a, int *b,
    int *c, int n) {
  for (i=0; i<n/4; i+=4)
    a[0..3] = b[0..3] + c[0..3];
}
```

下图中展示了 SLP（superword-level parallelism）向量化原理。

```
void foo(int a1, int a2,
    int b1, int b2,
    int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}
```

→

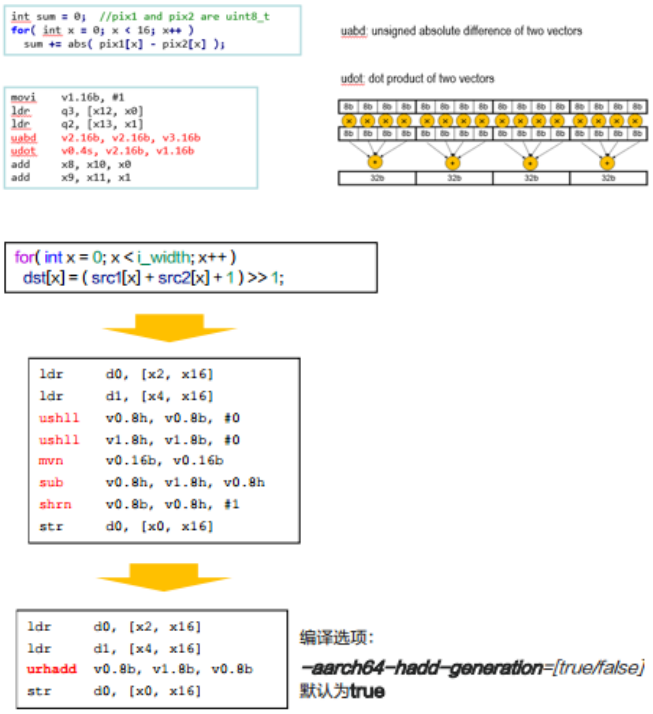
```
void foo(int a1, int a2,
    int b1, int b2,
    int *A) {
  A[0..3] = {a1,a2,a1,a2} * ({a1,a2,a1,a2} + {b1,b2,b1,b2});
}
```

基于以上原理，毕昇编译器通过自动矢量化，自动生成鲲鹏向量指令。以下两个示例具体展示了自动向量化优化的两个典型应用场景。

场景一：

场景二：

3) 自动调优 Autotuner: 自动调优是一种自动化的迭代过程，通过操作编译选项来优化给定程序，以实现



最佳性能。在过往的研究中，基于机器学习的编译器自动调优技术被证明比手工调优有更好的性能。

毕昇编译器的 AI 自动调优是由两个组件配合完成：毕昇编译器和 Autotuner 命令行工具。毕昇编译器是带有自动调优特性的编译器，配合 Autotuner 可以更细粒度地控制优化。Autotuner 是一个命令行工具，需要与毕昇编译器一起使用。它管理搜索空间的生成和参数操作，并驱动整个调优过程。

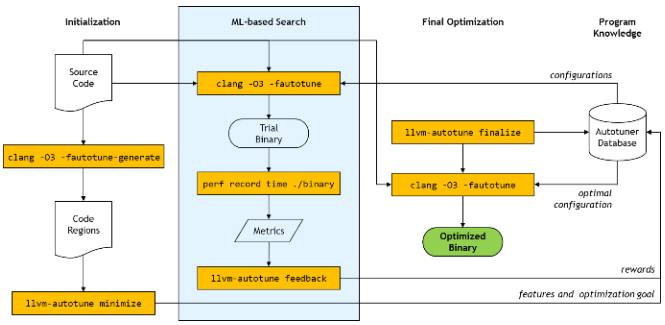


图 10: 毕昇自动调优 Autotuner 基本架构

毕昇自动调优 Autotuner 引入基于 ML 的自动搜索技术（ML-based Search），其关键技术点有：

1) 知识库 (Autotuner Database)：根据静动态分析信息，建立知识库，支持决策系统进行优化；
2) 优化决策系统 (Optimal configuration)：根据热点和性能评估信息、知识库信息，综合考虑确定优化措施；
3) 热点标记和性能评价：热点标记，瓶颈检测，性能评估；
4) 查找驱动 (Feedback)：将优化决策系统反馈的优化建议，反馈给编译器执行优化措施。

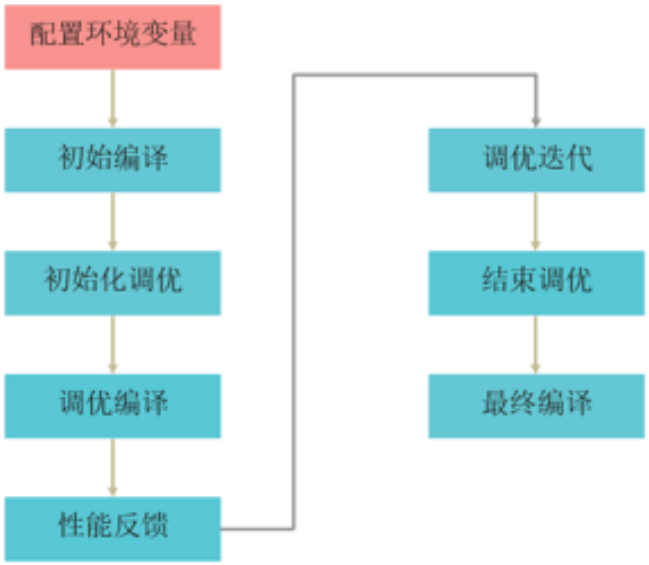


图 11: 毕昇自动调优 Autotuner 流程图

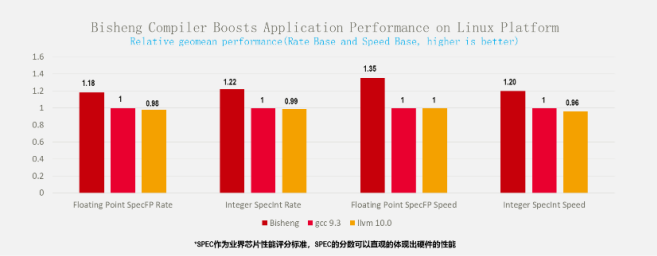
自动调优的关键流程如下：

1) 使用环境变量 AUTOTUNE_DATADIR 指定与调优相关数据的存放位置；
2) 添加毕昇编译器选项-fautotune-generate，完成初始编译，生成调优机会；
3) 运行 llvm-autotune 命令，初始化调优任务。生成最初的编译配置文件供下一次编译使用；
4) 添加毕昇编译器选项-fautotune，读取当前 AUTOTUNE_DATADIR 中的配置并编译；
5) 运行程序，并根据自身需求获取性能数字，使用 llvm-autotune feedback 反馈；
6) 根据用户设定的迭代次数，重复调优编译和性能反馈步骤进行调优迭代；
7) 经过多次迭代后，可选择终止调优，并保存最优的配置文件；
8) 使用上一步得到的最优配置文件，进行最后编译。

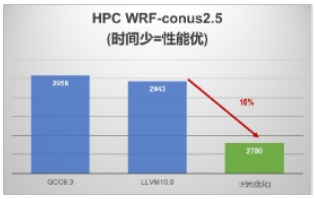
C. 毕昇编译器优化效果

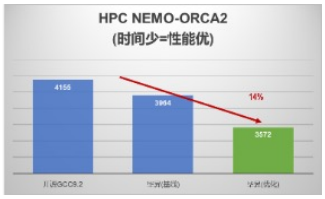
通过与鲲鹏芯片协同，毕昇编译器充分发挥芯片性能，提升鲲鹏硬件平台上业务的性能体验。本小节展示了毕昇编译器在不同 Benchmark 和不同应用场景下的实验结果。

1) 业界 CPU Benchmark 跑分: 下图展示了在鲲鹏上平台上，测试程序使用毕昇编译器，gcc 9.3 和 llvm 10.0 三个编译器优化后的相对性能。实验结果清晰地显示毕昇编译器的优化效果优于其余两个编译器，且将 SPEC 2017 性能平均提升 20% 以上。



2) HPC 典型应用性能提升: HPC 场景典型应用优化，结合鲲鹏芯片特性，通过毕昇编译器优化技术提升 HPC 应用的性能体验。





IV. 毕昇编译器未来创新演进方向

毕昇编译器将通过基础理论创新，向下软硬协同架构创新，向上业务场景精准调优，进一步全方位的技术创新。重要研究的方向有:

1: **程序持续优化**（Continuous Program Optimization，CPO）: 编译优化已不再局限在开发阶段，学术界和工业界均在探索高性能运行态优化技术，未来全流程持续优化将会是性能优化的重要布局方向。比如: IBM CPO 布局低成本的程序运行时性能监控技术，通过运行态程序行为变化分析，挖掘静态编译无法精确预测和优化机会点 [30]，还有米兰理工的 libVC 动态实时编译框架，从开发态的离线编译优化，拓展到运行态的在线编译优化，打通软件全生命周期性能持续优化能力 [31]，这些布局技术的角度虽然有所不同，最终通过构建持续优化，实现程序实时全生命周期的优化。

2: **AI 辅助优化**（AI for Compiler）: 人工智能在编译优化的应用目前已被大公司广泛研究与应用，未来借助 AI 辅助的编译优化会是编译优化重要发力方向。比如: Intel 和 UC Berkley 联合研究-深度强化学习辅助循环自动向量化，使用 deep reinforcement learning(DRL) 来对循环向量化进行优化决策，辅助循环向量化自动化。通过稳定的预测模型可直接嵌入到编译的自动向量化阶段，得到较为优化的向量化结果 [32] ；同样，Google 也提出了 AI 辅助 inling 优化策略，采用机器学习模型替代 opt inliner 的启发式算法，决策是否做 inline，实现更精细化的优化 [33]。

3: **SoC 级编译器**: SoC 级架构芯片已成为业界芯片公司发展的主要趋势，未来 SoC 级编译器可能会成为新的研究方向，比如: 苹果的 M1 芯片, PC 机处理器做成了手机 SoC，N 合一芯片，Intel Alder Lake，单一、高度可扩展的 SoC 架构，其中涉及统一内存架构（UMA 架构）、混合设计 SoC 架构（能效核和性能核）等新技术，需要布局探索配套的编译技术。

4: **超级优化器**（Super optimizers）: 学术界一直在尝试拓展编译器的基础理论研究，近几年超级优化器有从学术界到产业界加速应用落地的趋势，超级优化器将优化问题转换为空间搜索问题，通过探索新理论算法对程序特征序列 (指令集 (ISA) /中间表示 (IR)) 的有限搜索空间进行详尽搜索来尝试找到能够提供等效输出的最优程序。比如。斯坦福的 STOKE[34]、MTK 的 Souper[35] 等技术，已应用到 LLVM 编译器中。

V. 总结

毕昇编译器针对不同的芯片，不同的场景，不同的应用特点，使用不同的编译优化手段构建关键竞争力，在编译优化的过程中权衡考虑代价与收益，综合考虑性能收益，代码体积，编译时间，可调试性等多方面因素，通过软件与硬件的协同优化，充分发挥硬件极致算力。

参考文献

[1] Melvin E. Conway: Design of a separable transition-diagram compiler. Commun. ACM 6(7): 396-408 (1963)
[2] Philip M. Lewis II, Richard Edwin Stearns: Syntax-Directed Transduction. J. ACM 15(3): 465-488 (1968)

[3] Donald E. Knuth: On the Translation of Languages from Left to Right. Inf. Control. 8(6): 607-639 (1965)
[4] Allen, F. E.: Control flow analysis. SIGPLAN Notice 5(7), 1-19, 1970.
[5] Gary A. Kildall: A Unified Approach to Global Program Optimization. POPL 1973: 194-206
[6] Barry K. Rosen: High-Level Data Flow Analysis. Commun. ACM 20(10): 712-724 (1977)
[7] G.J. Chaitin, Register allocation and spilling via graph color-ing, United States Patent 4,571,678, February 1986.
[8] G.J. Chaitin, Register allocation and spilling via graph color-ing, SIGPLAN Not. 17 (6) (1982) 98–105. Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.
[9] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, P. Tu, Register promotion by sparse partial redundancy elimination of loads and stores, SIGPLAN Not. 33 (5) (1998) 26 – 37.Proceedings of the ACM SIGPLAN ' 98 Conference on Programming Language Design and Implementation.
[10] A.V.S. Sastry, R.D.C. Ju, A new algorithm for scalar register promotion based on SSA form, SIGPLAN Not. 33 (5) (1998) 15 – 25. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.
[11] Keith D. Cooper, Philip J. Schielke: Non-local Instruction Scheduling with Limited Code Growth. LCTES 1998: 193-207
[12] Jack L. Lo, Susan J. Eggers: Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism. PLDI 1995: 151-162
[13] Edward S. Lowry, C. W. Medlock: Object code optimization. Commun. ACM 12(1): 13-22 (1969)
[14] Randolph G. Scarborough, Harwood G. Kolsky: Improved Optimization of FORTRAN Object Programs. IBM J. Res. Dev. 24(6): 660-676 (1980)
[15] R. G. G. Cattell, Joseph M. Newcomer, Bruce W. Leverett: Code generation in a machine-independent compiler. SIGPLAN Symposium on Compiler Construction 1979: 65-75
[16] Martin Richards: The Portability of the BCPL Compiler. Softw. Pract. Exp. 1(2): 135-146 (1971)
[17] Marc A. Auslander, Martin Hopkins: An Overview of the PL.8 Compiler. SIGPLAN Symposium on Compiler Construction 1982: 22-31
[18] John Cocke, Peter W. Markstein: Measurement of Programming Improvement Algorithms. IFIP Congress 1980: 221-228
[19] Mark Scott Johnson, Terrence C. Miller: Effectiveness of a machine-level, global optimizer. SIGPLAN Symposium on Compiler Construction 1986: 99-108
[20] John Hennessy , David Patterson, "A New Golden Age for Computer Architecture" Stanford and UC Berkeley 13 June 2018
[21] Chris Lattner, " The Golden Age of Compilers" ASPLOS 2021 April 19, 2021
[22] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, Ion Stoica, "neuro vectorizer end-to-end vectorization with deep reinforcement learning".