第 3 章　并行编译器循环优化

循环优化是提升程序执行效率、降低控制流指令开销的关键技术。本章的主要工作集中在三个方面：一是利用 VECTOR DSP 架构中专用的 LOOP 指令实现零开销循环控制；二是针对芯片架构实现了循环分支重定位技术，解决了复杂分支跳转对控制流准确性的影响；三是探索了适应于 VLIW 特性的架构软件流水线优化。与前人工作相比，本章创新性地结合硬件循环指令特性和芯片架构进行优化，从循环结构、分支重定位到软件流水线构建，构成了一套全方位的循环编译优化方法，针对 VECTOR DSP 应用中的循环场景提供了坚实的技术支撑。

3.1　LLVM 循环体结构

在 LLVM 架构编译器的目标代码生成流程中，中间代码通常由多个函数（Function）构成，每个函数进一步划分为若干基本块（Basic Block）。基本块是 LLVM 中进行代码生成与优化的最小单位，由一组顺序执行的指令组成，具有单一的入口和出口，这种特性使其便于进行控制流分析与优化。在控制流中，每个基本块的最后一条指令通常为控制流指令（如条件跳转或无条件跳转），这些指令决定了后续执行路径，从而构成复杂的程序逻辑结构。

在处理循环结构时，LLVM 通过一系列标准化的基本块类型来表示循环体，这些基本块在控制流图中形成如下关键组件，具体结构如图 3.1：

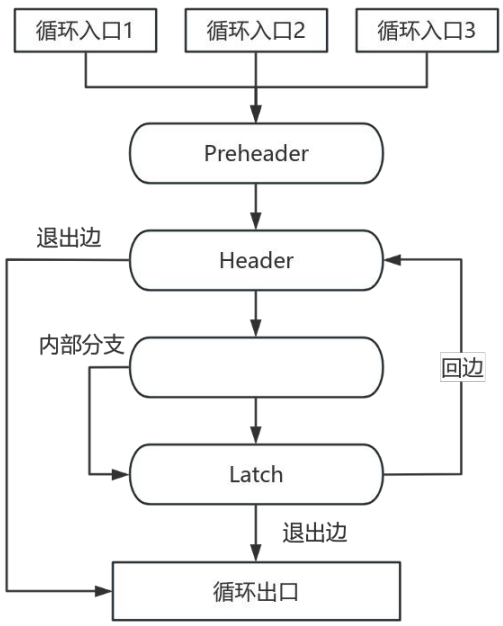

图 3.1　LLVM 循环体的基本块结构

1、前置块（Preheader Block）：前置块是循环开始前执行的基本块，通常用于初始化循环变量并将控制流引导至循环头块。前置块的引入对于优化具有重要意义，例如循环不变代码的外提（Loop-Invariant Code Motion, LICM）[65]，可以将不需要重复计算的指令移至前置块中，从而减少循环体中的重复开销。

2、循环头块（Header Block）：循环头块是循环的入口基本块，通常包含循环条件的判断逻辑。根据判断结果，控制流会跳转至循环体块继续执行，或进入出口块结束循环。LLVM 将循环头块作为循环检测和优化的标识点，在循环展开（Loop Unrolling）等优化中起到重要作用。

3、循环体块（Body Block）：循环体块包含需要重复执行的指令集合，承担核心计算任务。在复杂循环中，循环体块可能由多个基本块构成。

4、闭合块（Latch Block）：闭合块负责更新循环控制变量，并将控制流重新导向循环头块以检查循环条件。

5、出口块（Exit Block）：出口块在循环条件不满足时执行，负责处理循环结束后的逻辑。

通过利用 LLVM 的优秀结构设计和模块化复用能力，可以对循环中的 Preheader、Header 以及 Latch 块进行深入分析与处理。这一机制不仅提升了芯片循环代码执行的性能，还能够更好地适配目标平台的特定硬件特性。

3.2　VECTOR DSP 硬件循环设计

3.2.1　循环类型

根据循环迭代次数的确定性，可以将循环分为以下三种主要类型[66]，下文分别对这三种循环类型的概念进行详细说明：

1、静态循环：静态循环的迭代次数在编译阶段即已固定，通常由明确的常量值或确定的表达式定义。这类循环的迭代次数不会受到运行时条件或外部数据的影响，因此编译器能够通过静态分析直接确定其执行次数。静态循环因其确定性和可预测性，便于在编译阶段进行一系列优化。

2、 半静态循环：半静态循环的迭代次数虽然在编译阶段无法解析为一个立即数，但在程序运行的某个阶段能够确定下来，通常是在循环体执行之前明确。

3、动态循环：动态循环的迭代次数在程序执行过程中动态变化，且无法在运行时开始前预先确定。这类循环的执行次数通常依赖于复杂的运行时条件。

VECTOR DSP 支持对静态循环和半静态循环进行循环编译优化，该架构支持以下功能特性：循环包含多个循环体的入口（entry point），多个循环体出口（exit point）以及循环体内部存在分支语句。

VECTOR DSP 循环编译优化主要集中于单层循环，这与数字信号处理器（DSP）的实际应用场景密切相关。在嵌套循环的场景中，计算强度最大的部分通常集中在最内层循环体，这里也是执行效率的瓶颈所在。因此，编译器优先针对最内层循环进行优化。展望未来，随着 VECTOR DSP 硬件架构中引入多个循环计数器的支持，循环编译优化的能力将进一步扩展至多层嵌套循环。这种改进将为复杂的数字信号处理应用带来更高的性能，尤其是在大规模数据处理和实时系统的场景中。

3.2.2　硬件循环指令设计

硬件循环(Hardware Loop)也称为零开销循环，硬件循环的迭代控制不再通过程序中的汇编指令来实现，而是通过专用的硬件单元来实现控制，从而节省循环每次迭代所需要的指令开销。

在 VECTOR DSP 架构中，为实现高效的循环编译优化，设计了一条专门的指令 LOOP，该指令的定义通过 TableGen 工具描述，代码如下：

```
class LoopBase<bits<3> typeop, bits<3> op, bits<4> inner_op, string instr_asm
>:   F_Reg1_Imm16<typeop,op,inner_op, (outs), (ins CPURegs:$ra,
          brtarget:$imm16),!strconcat(instr_asm, "\t$ra $imm16"),
          [], ControlFlow_Itin> {
      let Inst{3-0} = inner_op;
      let isBarrier = 1;
      let hasDelaySlot = 1;
      let isTerminator = 1;
      let hasSideEffects=0;
}

def LOOP : LoopBase<4, 5, 0b0000, "loop">, Sched<[WriteCOND,
                                                  ReadCOND, ReadCOND]>;
```

LOOP 指令包含两个主要参数：

循环迭代计数：第一个参数是一个通用寄存器，用于存储循环体的执行次数。此寄存器的值在指令开始时加载，作为循环迭代的上限控制器。

循环体地址偏移：第二个参数是一个地址标签，采用 16 位的二进制立即数表示，记录循环体相对于 LOOP 指令程序计数器（PC）的偏移量。具体而言，该偏移量指向循环体内最后一条 VLIW 指令的位置，为实现硬件循环提供了精准的跳转目标。编译过程中由伪指令 ENDLOOP 标识。

为了在 VECTOR DSP 流水线中实现零开销循环，在 IF2（取指）阶段设计了三个专用控制寄存器，分别用于存储循环起始地址（StartLoop）、循环终止地址（EndLoop）以及循环迭代计数（LoopCount）。这些寄存器的值由 EX 阶段通过解析 LOOP 指令的参数计算得出，并随后写入 IF2 阶段的控制逻辑中。

在循环执行过程中，IF2 阶段每从指令存储器中取出一条 VLIW 指令并递增程序计数器（PC）后，会检测当前 PC 值是否与 EndLoop 相等，以判断是否到达循环体的末尾。如果 PC 值与 EndLoop 匹配，流水线逻辑会递减 LoopCount 寄存器的值，并检查其是否小于零。若 LoopCount 小于零，则循环终止；否则，将 FetchAddr 寄存器重置为 StartLoop，从而确保在下一周期内能够正确取回循环体的第一条指令，同时将 PC 寄存器同步重置，实现循环的自动迭代。

通过这一设计，VECTOR DSP 能够在不增加额外开销的情况下，高效地支持循环体的重复执行，从而提升代码性能和硬件资源利用率，同时为深度优化数字信号处理任务奠定了基础。

3.3　循环优化算法设计与实现

循环优化算法流程如图 3.2。

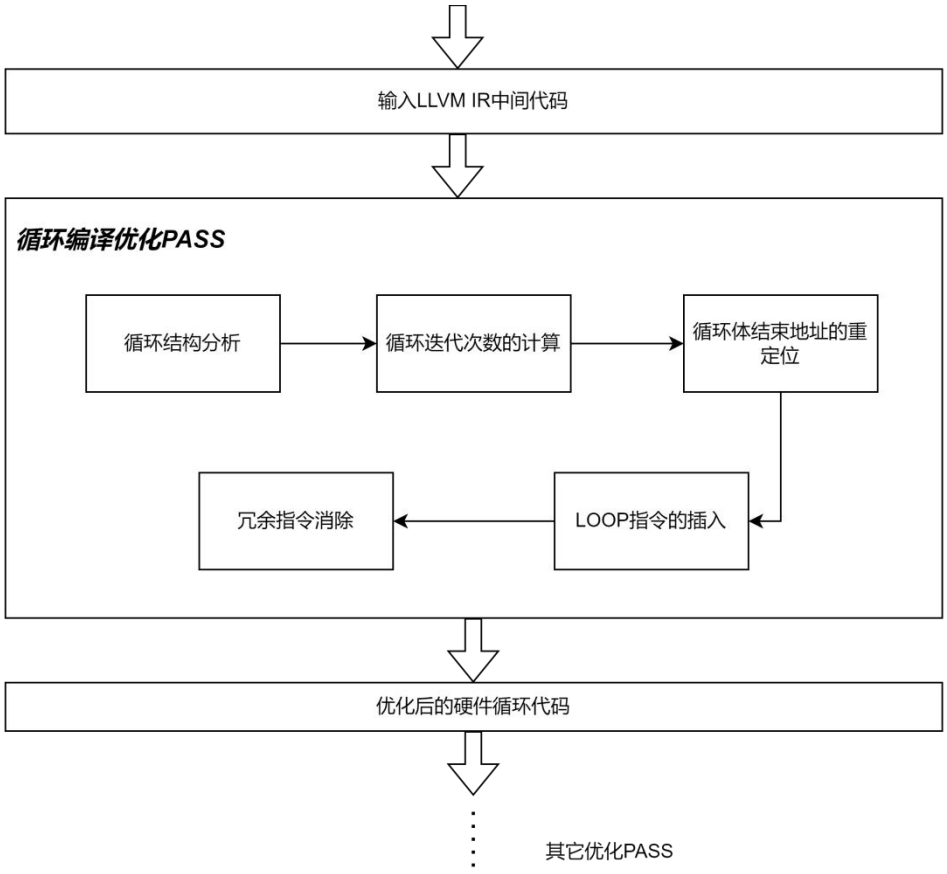

图 3.2　循环优化算法流程图

3.3.1　循环可优化性检查

循环编译优化目前支持静态循环与半静态循环且编译器优先针对最内层循环进行优化，故需要在应用循环编译优化时对循环结构进行检查，对符合上述要求的合法的循环进行优化。循环合法化检查算法伪代码如下所图 3.3。

Algorithm 1　convertToHardwareLoop

```
Input: MachineLoop L
Output: Boolean value indicating if the loop was converted to a hardware loop
1     Initialize Changed ← false
2     // Recursively process nested loops
3     for each loop I in L do
4           Changed ← Changed OR ConvertToHardwareLoop(I)
5     end for
6
7     if Changed == true then
8           return Changed
9     end if
10
11    if containsInvalidInstruction(L) then
12          return false   // Check if there are invalid instructions in the loop
13    end if
14    if fixupInductionVariable(L) == false then
15          return false   // Check if the induction variable is valid
16    end if
17
18    TripCount ← getLoopTripCount(L, ...)   // Check if loop iteration count is determinable
19    if TripCount == null then
20          return false
21    end if
22    return true
```

图 3.3　循环可优化性检查算法伪代码

编译器针对 LLVM IR 中的每个循环体，采用递归方式进行逐层检查与优化。具体来说，首先定位嵌套结构中最内层的循环，优先尝试对其进行分析和转换。只有当前层级的所有循环均无法满足优化条件时，才会递归回退至上一层循环并对其进行相应处理。

在每个循环体的检查过程中，编译器会验证是否可以提前确定循环迭代次数以及其是否包含非法指令，例如已经存在的 LOOP 指令或函数调用等可能干扰循环可优化性的语句。若检测到不符合条件的语句，当前循环将被直接跳过，编译器继续处理其他循环体，以确保整体转换流程的高效性与正确性。

3.3.2　LOOP 指令生成

LOOP 指令的生成主要分为四个步骤：1、循环结构分析 2、循环迭代次数的计算 3、循环体结束地址的重定位 4、LOOP 指令的插入

(1)循环结构分析

LLVM 提供了功能丰富且易于使用的接口，为开发者分析循环结构提供了极大便利，帮助开发者深入理解循环的控制流和数据流，从而提取与循环相关的关键信息,这些关键信息构成了循环编译优化的基础。使用 LLVM 提供的循环分析函数，可以高效地获取循环编译优化所需的核心信息，如循环计数、循环体的起始地址和结束地址。这些信息是实现循环编译优化逻辑构建的重要依据，能够为编译器生成高效的目标代码提供支持。

函数 MachineBasicBlock *Header = L->getHeader(); 返回了循环的头块，即循环结构的入口点。这一接口用于确定循环的开始地址，为后续硬件循环的构建提供关键基础。

函数 MachineBasicBlock *Preheader = L->getLoopPreheader(); 则用于获取循环的前置块。前置块是在循环正式执行之前的准备区域，为硬件循环指令（如 LOOP 指令）的插入提供了一个合适的位置，确保循环的初始化和控制流逻辑能够正确执行。

最后，函数 MachineBasicBlock *Latch = L->getLoopLatch(); 返回了循环的闭合块。闭合块是循环控制变量更新的位置，通常用于判断是否需要跳转回循环头块或结束循环。该接口为循环结束条件的处理和跳转逻辑的实现提供了关键支持。

通过以上接口，编译器能够高效地提取和利用 LLVM 循环结构中的核心信息，从而构建和优化循环逻辑，为硬件指令的生成和循环编译优化提供了强大的技术支撑。

(2)循环迭代次数的计算

循环迭代次数的计算是循环编译优化算法正确实现的关键环节。为了精确地计算循环的迭代次数，通常需要满足两个前置条件。首先，必须明确循环迭代变量。其次，需要确定这些迭代变量的上下界。这两个条件为后续的循环优化提供了基础，确保了编译器能够对循环进行有效的优化。

在 LLVM 编译框架中，循环体通常以若干条 phi 指令作为起始。phi 指令的格式如公式 3.1 所示：

$$\%var0 = phi32\left[\%value0,\%block0\right],\left[\%value1,\%bl\,ock1\right],\cdots \tag{3.1}$$

其中每个参数是一个包含数值和基本块标签的二元组。具体来说，phi 指令

的作用是将循环变量与其所在基本块的前驱块关联起来。例如，如果控制流从基本块 block0 跳转到当前基本块，则变量 var0 将被赋值为 value0；如果控制流从基本块 block1 跳转过来，则 var0 将被赋值为 value1，以此类推。这样的结构使得编译器能够在不同的控制流路径下正确地跟踪和更新循环变量。

在 LLVM 中，循环迭代变量的定义通常通过 phi 指令实现。在循环的头部（Header）基本块中，编译器会分析每一条 phi 指令，检查其各个参数的候选值，并分析这些参数与前驱基本块的关系。具体来说，当编译器发现某个 phi 指令的参数来自循环体外的一个基本块和来自循环体内的 Latch 块，并且该 phi 指令定义的变量在 Latch 块中执行了递增（或递减）操作、比较操作，并最终通过 br 语句跳转回循环头部时，就可以确定该变量为循环迭代变量。根据这一模式，编译器能够识别出循环的控制结构，进而为后续的优化过程奠定基础。

进一步分析，当编译器识别出循环迭代变量后，它还会通过检查该变量的递增步长以及与边界值的比较操作，来推导出该变量的上下界。例如，在 Latch 块中，编译器可能会看到类似于%9 = add nuw nsw i32 %5, 1 的递增操作，意味着该变量每次递增 1，步长为 1；同时，通过类似%10 = icmp eq i32 %9, 3 的比较操作，编译器可以确定循环迭代的终止条件，即迭代变量的上界为 3。基于这些信息，编译器可以计算出循环的迭代次数，通常通过公式 3.2 来得出，并将结果保存在 LOOP 指令的第一个参数对应的寄存器中。这一过程确保了编译器能够准确计算循环的迭代次数，从而为进一步的循环编译优化提供支持，优化程序执行效率。

以图 3.4 中的部分 LLVM IR 代码为例，在 LLVM IR 中，phi 指令用于定义循环迭代变量 %5。该变量的初始值为 0，并且在每次迭代时，它的值会根据前一次的值递增。当从 Latch 块跳转到当前块时，%5 的值被更新为变量 %9 的值。具体而言，指令 %9 = add nuw nsw i32 %5, 1 表示每次迭代时，变量 %5 会增加 1。而比较操作，icmp eq i32 %9, 3 指令用于判断 %9 是否已达到 3。如果是，则跳出循环；否则，继续执行下一次迭代。通过这样的递增操作和边界值判断，则能够在编译过程中准确地计算出循环的迭代次数为 3。

```
4:                                                ; preds = %4, %0
  %5 = phi i32 [ 0, %0 ], [ %9, %4 ]
  call void @llvm.dbg.value(metadata i32 %5, metadata !22, metadata !DIExpression()), !dbg !30
  %6 = getelementptr inbounds [3 x i32], [3 x i32]* @Array, i32 0, i32 %5, !dbg !33
  %7 = load i32, i32* %6, align 4, !dbg !35, !tbaa !26
  %8 = add nsw i32 %7, 1, !dbg !35
  store i32 %8, i32* %6, align 4, !dbg !35, !tbaa !26
  %9 = add nuw nsw i32 %5, 1, !dbg !36
  call void @llvm.dbg.value(metadata i32 %9, metadata !22, metadata !DIExpression()), !dbg !30
  %10 = icmp eq i32 %9, 3, !dbg !37
  br i1 %10, label %3, label %4, !dbg !31, !llvm.loop !38
```

图 3.4　循环体 LLVM IR 代码

(3)循环体结束地址的重定位

循环结束地址是 LOOP 指令的第二个参数，通常以闭合块（Latch Block）中的第一条终结指令来确定。该地址标识循环体的最后一条指令，用于判断循环是否结束或是否需要跳转回循环头块以开始下一次迭代。在循环编译优化过程中使用伪指令 ENDLOOP 来对结束地址进行标识，伪指令 ENDLOOP 的地址被重定位为循环体的结束地址，以确保控制流的正确性。闭合块中的终结指令在循环控制中起到了关键作用，通过其运算逻辑，硬件能够准确地执行循环体的跳转与退出操作。这种方法不仅确保了循环体在符合条件下能够正常终止，还提升了循环体的执行效率与鲁棒性。

(4)LOOP 指令的生成

LOOP 指令是一种专为循环执行设计的硬件控制指令，旨在实现对循环过程的精准、高效管理。通过该指令，芯片能够明确循环的迭代次数以及循环体的执行范围，从而避免传统循环实现中常见的条件判断和跳转操作所带来的额外指令开销。作为循环优化的核心控制机制，LOOP 指令的加入不仅简化了循环的控制逻辑，还显著提升了循环体的执行效率，尤其在高频迭代的场景中，其性能优势表现得尤为突出。

具体而言，在循环优化指令生成阶段选择将 LOOP 指令插入到前置块（Preheader）当前指令序列的最后一条指令之后。这种插入方式的主要优点在于，它为循环优化的初始化和后续循环体内分支重定位调整提供了一个明确的锚点。

通过将 LOOP 指令置于前置块的末尾，既可以保证循环体外部指令在硬件层面按预期执行，又避免对循环体内部基本块结构造成干扰。这种设计使循环体的控制更加高效和清晰，便于后续编译优化流程的进行，同时为目标流水线性能优化提供了支持。

3.3.3　冗余指令消除

在传统的循环结构中，为了保证循环迭代变量的正确更新和循环条件的准确判断，编译器通常需要插入多条额外的计算以及控制指令。例如，当未启用循环编译优化时，VECTOR DSP 芯片架构中每次循环迭代涉及五条附加指令的开销，包括更新循环迭代变量的计算指令、判断循环条件的分支指令、跳转指令以及跳转指令所引发的延迟槽开销。延迟槽是由于流水线架构的特性而额外引入的指令填充开销，这进一步加剧了性能损失。这些额外的控制指令会显著增加循环的执行成本，尤其是在高频迭代场景中，对性能产生不利影响，并且会增加指令存储的压力。

通过引入循环编译优化，这些由循环控制引起的冗余指令开销可以被显著消除。LOOP 指令在硬件级别直接进行循环迭代的控制，能够一次性实现循环计数的初始化、循环条件的隐式判断和循环跳转操作。因此，在循环编译优化的支持下，传统循环控制所需的显式条件判断和跳转指令变得不再必要，从而大幅简化了循环的控制逻辑。

在优化过程中，编译器会对循环结构进行详细分析，在编译器完成循环编译优化 Pass 后，需要将循环体中原有的冗余指令进行移除，这些指令通常位于循环头块（Header Block）和闭合块（Latch Block），具体位置根据硬件特性和指令依赖关系确定。以图 3.5 为例，LOOP 指令生成后的 LLVM IR 代码如图中(b),则(a)图中所示的与控制循环迭代相关的指令将变为冗余指令，可对(a)图中的迭代变量递增指令(addi)、比较指令(gti)、跳转指令(jnc)及其延迟槽(nop)进行移除，这种优化流程不仅减少了循环体中指令数量，提高了执行效率，同时还降低了最终生成代码的尺寸。

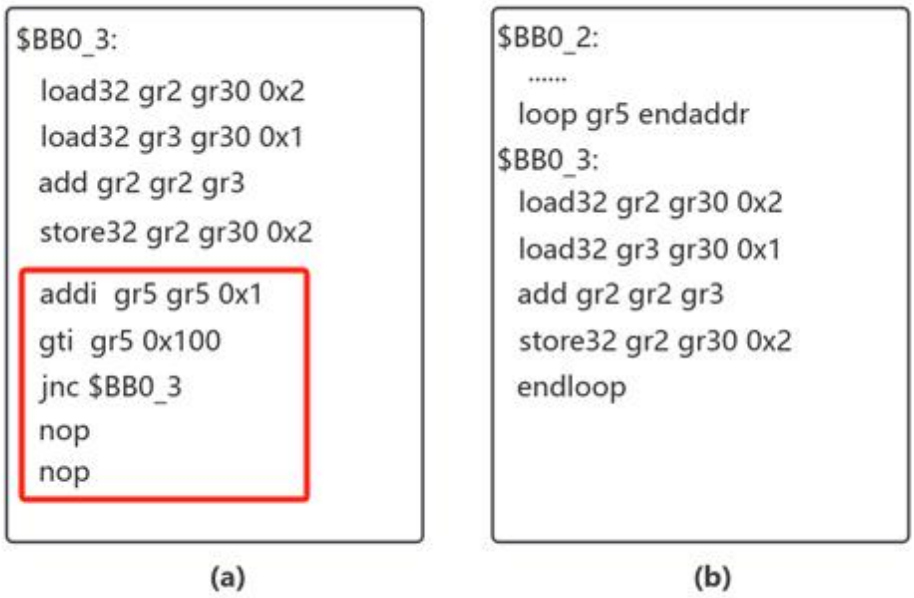

图 3.5　循环编译优化前后 LLVM IR 代码

3.4　目标架构特性相关循环优化

目标架构特性相关的循环优化是循环编译优化的重要部分，具体包括循环内部分支重定位和软件流水线优化。循环内部分支重定位通过减少分支指令的执行频率，有效降低控制流开销；软件流水线优化则通过重叠不同循环迭代的指令执行，充分利用并行处理能力。这些技术与芯片的并行计算特性和灵活指令集相结合，显著提高了循环执行效率，为高性能嵌入式应用提供了强有力的支持。

3.4.1　循环分支重定位

在实际代码中，循环结构往往不仅包含简单的计算语句，还可能包含分支跳转等复杂的控制流语句。当循环运行到某一分支时，该分支可能直接终止当前循环的执行（例如，遇到某种条件后跳出本次循环的后续计算），并通过跳转指令跳转到 ENDLOOP 的位置。然而，ENDLOOP 是一条伪指令，它会在后续编译过程中被消除。因此，为了保证循环的执行逻辑正确无误，需要对循环内的分支跳转地址进行适当的重定位，效果如图 3.6。

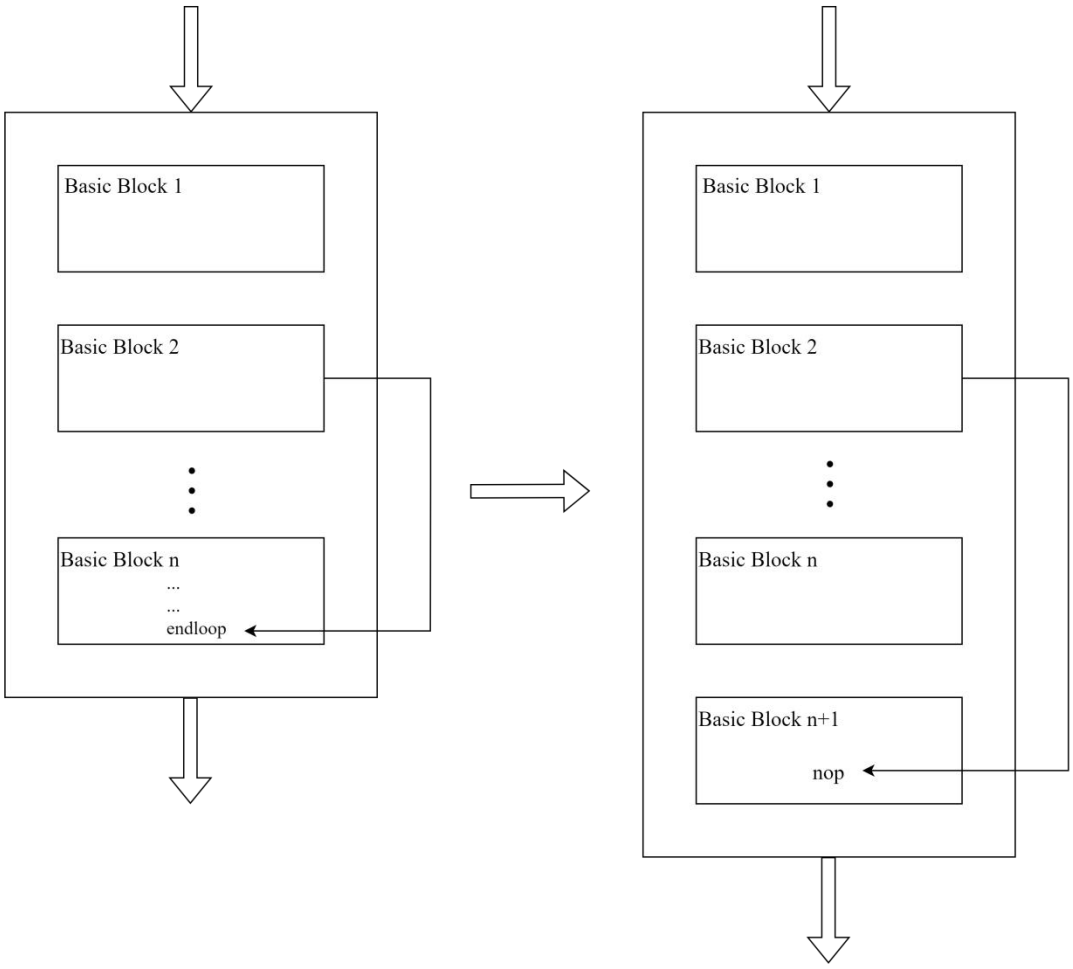

图 3.6　循环结束地址修正

为此，编译器在完成循环结束地址的修正后（通过对 Latch 块进行拆分），还需要进一步检查 Latch 块是否是循环体中某些分支指令的跳转目标地址。如果存在这种情况，编译器会新建一个 Landing 块（Landing Block），并将其插入到最终的 Latch 块之前。随后，原本跳转到 ENDLOOP 的分支指令将被重定向至新建的 Landing 块，从而保证控制流的正确性。Landing 块在此处起到了缓冲和桥接作用，使分支跳转能够正常完成而不会破坏循环的控制逻辑，循环编译优化的最终汇编代码如图 3.7。

```
$BB0_2:
  ......
  loop gr5 $BB0_4
$BB0_3:
  load32 gr2 gr30 0x2
  load32 gr3 gr30 0x1
  add gr2 gr2 gr3
  store32 gr2 gr30 0x2
  gti  gr2 gr6
  jc  $BB0_7 //及其延迟槽
$BB0_7:  //landing block
  nop
$BB0_4:
  addi gr7 gr7 5
```

图 3.7　循环内部分支重定位后的汇编代码

虽然这种处理方式会导致循环体每次迭代的执行时间增加一个流水线周期，因为新增的 Landing 块会引入额外的跳转操作。然而，这类特殊的内部分支情况在实际代码中相对少见，因此即使存在此类特殊情况，通过移除控制循环迭代变量的自增、比较以及跳转指令及其延迟槽带来的开销，循环体总体的执行时间依然得到了显著的优化。换言之，尽管新增的 Landing 块可能略微增加单次迭代的延迟，但硬件循环通过优化其他冗余指令，抵消了这一影响，最终显著提升循环的整体执行效率。

这种处理方式体现了编译器在复杂循环场景中的灵活性，能够兼顾特殊分支情况的正确性与性能优化之间的平衡，同时确保循环编译优化的功能实现与运行效率的最大化。

3.4.2　软件流水线优化

软件流水线技术是指令调度的一种，它的思想是通过对循环的连续迭代来探索指令的并行性，尽可能的使更多的指令在循环中实现并行。在软件流水线技术中，循环的一次迭代被分为多个阶段。这样的好处是能使得连续迭代的能够处于不同阶段实现重叠执行，具有更好的并行性。针对 DSP 架构软件流水线算法流程如下：

首先计算初始化间隔的最小值（Minimum Initiation Interval, MII），MII 是软件流水线优化的关键参数，指循环迭代间允许的最小时间间隔，用于确保指令调度在硬件资源和依赖关系约束下实现高效并行执行。MII 的作用在于为初始化间隔（II）设定下限，避免尝试过小的 II 值导致计算复杂度过高或资源冲突，从

而加速调度过程并优化处理器性能。在 VECTOR DSP 芯片的 VLIW 架构中，MII 确保多指令发射槽的并行性与循环执行效率最大化，减少流水线停顿。

MII 的计算受最小资源初始化间隔（ResII）和最小迭代初始化间隔（RecII）两个参数约束。其计算公式为：

其中：

$$\mathrm{MII} = \max\left(\mathrm{ResII}, \mathrm{RecII}\right) \tag{3.2}$$

ResII（最小资源初始化间隔）基于一轮迭代所需计算资源的总量与处理器可用资源综合计算：

$$\mathrm{ResII} = \max\left(\frac{\text{一次迭代中需要的i类资源总和}}{\text{处理器上i类资源综合}}\right), \mathrm{i} \in \text{处理器上资源种类} \tag{3.2}$$

RecII（最小迭代初始化间隔）表示循环中环路完成一次迭代所需的最小间隔：

$$\mathrm{RecII} = \max\left(\frac{\text{回路i的延迟总和}}{\text{回路i的距离总和}}\right), \forall \mathrm{i} \in \text{相关回路} \tag{3.4}$$

回路 i 的距离总和为其节点总数。通过上述计算，MII 为依赖图分析和调度提供基础。

之后对节点进行排序并进行调度，排序的目标是提升关键路径节点优先级，减少最大寄存器使用数。基于 VECTOR DSP 的 VLIW 特性，设计启发式排序：优先将可移动时间（MOB）较大的节点排在后面，可移动时间指的是节点在最早可能调度时间（ASAP）和最晚可能调度时间（ALAP）之间的时间范围。节点被放置在前置节点和后续节点之间，并尽量靠近前置节点，以减少节点的生存周期（lifetime），从而降低对寄存器的压力，优化资源利用；递归时，优先调度 RecMII 最高的节点。算法分两步排序：首先按优先级降序分组部分排序；然后通过双向遍历（从底部向上、顶部向下），优先选择可移动时间（MOB）最小的节点，若 MOB 相等则优先选择高度（H）或深度（D）较高的节点，生成高效指令序列。

由于要考虑到依赖关系的限制，节点排序后的指令节点在实际插入时间槽时并不能直接按照顺序进行插入。在最后通过调度算法实现调度操作时，定义了 ESTART 表示该指令节点可进行调度的最早周期数，LSTART 表示该节点可进行调度的最晚周期数。调度算法根据依赖计算指令最早（ESTART）和最晚（LSTART）周期，按范围插入时间表：有前置/后续节点时插在 [ESTART, LSTART] 或其子集，无依赖时从理论最早时间开始；若无槽位，则增大初始化间隔，对其重新进

行调度，最终实现软件流水线优化。

3.5　本章小结

本章从 LLVM 循环结构分析入手，探讨了 VECTOR DSP 芯片在循环编译优化方面的设计与实现策略。首先，深入剖析了循环中各基本块在控制流和数据流中的作用。接着，介绍了芯片通过 LOOP 指令及控制寄存器实现零开销循环控制，详细描述了针对循环结构的编译优化算法，包括合法化检查、LOOP 指令生成和冗余指令消除。此外，结合 VECTOR DSP 的架构特点，本章还探讨了循环内部分支重定位和软件流水线优化等技术，进一步挖掘了芯片在数字信号处理领域的潜力。本章设计实现的循环编译优化技术已通过自动化测试框架的回归测试模块验证，其性能测试与分析将在第五章详细呈现。