通过合并 IR 指令消除分支

Yuze Li*
lyuze@vt.edu 弗吉尼亚理工学院 美国弗吉尼亚州布莱克斯堡

Srinivasan Ramachandra Sharma*
srinivasanr@vt.edu 弗吉尼亚理工学院 美国弗吉尼亚州布莱克斯堡

Charitha Saumya
charitha.saumya.gusthinna.waduge@intel.com
英特尔公司 美国加利福尼亚州圣克拉拉

Ali R. Butt
butta@vt.edu 弗吉尼亚理工学院 美国弗吉尼亚州布莱克斯堡

Kirshanthan Sundararajah
kirshanthans@vt.edu 弗吉尼亚理工学院 美国弗吉尼亚州布莱克斯堡

arXiv:2512.22390v1 [cs.PL] 26 Dec 2025

摘要

分支误预测会给现代处理器带来灾难性的性能惩罚，导致性能下降。尽管存在硬件预测器和基于剖析的技术，但具有不规则模式的数据相关分支仍然难以处理。传统的 if 转换通过软件谓词化消除分支，但在 x86 等架构上存在局限。对于包含内存指令的路径，它往往无法奏效；而若对较大的分支主体进行完整推测执行，则会造成过高的指令开销。

本文提出了融合中间表示指令（MERIT），这是一种编译器变换，通过在中间表示指令级对齐并融合分歧路径中的相似操作来消除分支。MERIT 注意到，分歧路径通常会执行结构相似但操作数不同的操作，因此采用序列比对来发现融合机遇，并通过安全的操作数级保护确保语义正确性，而无需硬件谓词化。MERIT 以 LLVM pass 的形式实现，并在来自四个基准测试套件的 102 个程序上进行评估；与硬件分支预测器相比，其几何平均加速比达到 10.9%，峰值提升为 32×，展现了其在降低静态指令开销方面的有效性。

1 引言

现代超标量处理器依靠推测执行来维持较高的指令吞吐量。条件分支会引入关键的控制流冒险，打破这种线性流程，迫使处理器猜测正确的执行路径。误判分支路径可能导致严重的性能损失（IPC 损失最高可达 18% [17]）。

硬件分支预测已取得显著进展 [18,25–28], ，基于剖析的方法也能消除特定的棘手分支 [15, 30]。然而，对于具有不规则模式的数据相关分支，由于预测准确率与芯片面积之间存在权衡，它们仍然难以处理。

一种互补的基于软件的方法——*if* 转换，无需任何硬件改动即可彻底消除控制流冒险 [1–3, 16, 19, 22], ，尽管编译器难以判断哪些分支注定会被错误预测。编译器不再生成条件跳转，而是生成一条由谓词化指令构成的直线序列。if 转换在**机器级中间表示**上对分支进行操作：当所有控制相关于某个分支的指令都使用与该分支相同的条件进行谓词化时，就可以合法地移除该分支。然而，这种变换在 x86 上存在两个根本限制。首先，x86 缺少硬件谓词化支持。if 转换只能采用基于推测的方法，执行两条路径并在最终结果中进行选择。然而，它无法处理包含不安全内存操作的分支（从可能无效的地址加载，或条件存储），导致许多“可转换”的分支未能进行变换。其次，在 x86 上，即使变换成功，复制两条路径中所有操作所产生的开销也可能超过分支消除带来的收益，尤其是对于包含大量计算、结构相似的分支。

本文介绍**Mᴇlding IR Insᴛructions**（MERIT），这是一种编译器变换，与传统的 if 转换不同，它在目标无关中间表示级别上运行，而非目标相关的低层级。我们观察到，发散的控制流路径通常会执行结构相似的操作，主要区别在于其操作数。MERIT 不会推测性地执行整个路径，而是通过识别两条分支中相同的操作，使两侧的指令序列对齐。随后，这些对齐指令会合并为一条带有条件操作数选择的指令；而未对齐指令则会在与 MERIT 插入的冗余指令安全匹配（即保持正确性）后进行合并。

这种指令级方法相比传统的 if 转换有三大优势。**(1)通过语义分析安全地处理内存操作**：不同于硬件谓词化或基于推测执行的 if 转换，MERIT 会在中间表示级别进行语义分析，以推理

*贡献相同。

内存安全。在融合过程中，条件加载和存储会变为无条件加载和存储：通过分支条件正确保护用于选择地址和值的操作，从而确保结果正确。在中间表示级别进行这种语义推理，使 MERIT 能够变换传统上会被 if 转换跳过的分支。**(2) 通过中间表示指令融合降低静态指令开销**：MERIT 不会复制完整路径，而是合并结构相似的指令，因此生成的操作更少。当两个分支都计算 x = a +offset，但偏移量不同时，传统的 if 转换会生成四个操作（两次加法、两次选择），而 MERIT 只生成两个操作（一次选择、一次加法）。这能缩减代码体积、提高指令缓存利用率，并尽可能减少可能抵消分支消除收益的开销。**(3) 目标无关的中间表示级变换**：这种变换与架构无关，无论底层架构如何，均可应用。MERIT 会生成无分支的基于选择操作的代码，从而为后续优化创造条件。直线代码有助于改进指令调度、寄存器分配和向量化，而这些优化机会会受到控制流边界的限制。为考察性能收益，本研究聚焦于 x86 架构。

我们的主要贡献如下：

- MERIT：一种新颖的编译器变换，通过语义分析实现指令级对齐与融合，从而彻底消除分支，并解决传统 if 转换的根本局限。
- 使用冗余指令实现完全对齐：MERIT 改进了 Smith-Waterman 算法 [29]，以发现不同路径上的局部最优对齐，并插入安全的冗余指令来填补无法融合的间隙，从而实现完整的指令序列对齐。
- 正确性与安全性保证：MERIT 确保插入冗余指令后的数据流不会破坏原始程序的数据流。它利用语义分析确保内存访问安全，而传统的 if 转换无法做到这一点。
- 我们将 MERIT 实现为 LLVM pass，并使用来自四个基准测试套件的 102 个基准测试进行评估。与纯硬件推测相比，平均加速比达到 10.9%，峰值提升达 32×。

在本文余下部分中，我们首先介绍本项工作所需的背景知识（第 2 节）。接着，在第 3 节中，我们通过关键示例阐释指令级方法的动机，并在第 4 节中说明 MERIT 转换的设计。第 5 节介绍我们将其实现为 LLVM pass，以及如何将其与配置文件引导优化（PGO）集成；随后在第 6 节进行全面评估。最后，我们在第 7 节讨论局限性和未来工作，在第 8 节介绍相关研究，并在第 9 节总结全文。

2 背景

2.1 控制流冒险与分支预测

现代处理器采用深流水线和推测执行来实现较高的指令每周期。遇到条件分支时，处理器必须预测其方向，并从预测路径推测性地提取指令。预测正确时，流水线得以持续运行；预测错误则会导致流水线完全清空并重新启动，而且流水线越深，惩罚越大。数十年来，研究人员一直深入探索分支预测设计，以提高预测准确率。最先进的硬件预测器通常是类 TAGE [25–28], 基于感知器 [12, 13], , 或两者的组合 [11, 27]。这两类预测器各有优势：TAGE 能非常高效地利用有限的预测器存储空间，而基于感知器的预测器则能轻松结合不同类型的输入信息。近期研究人员利用配置文件引导优化（PGO）来提高预测准确率 [15, 30, 34]。然而，要实现较高的预测准确率，往往需要牺牲芯片面积。此外，重新设计硬件以适配新的编译器提示并不可扩展，因为这需要专门的指令集架构来支持软件提示，从而妨碍其在实际生产环境中的部署。

2.2 if 转换

if 转换是一种成熟的编译器技术，它通过将控制流转换为数据流来消除分支 [1–3, 16, 19, 22]。其目标是消除可能限制指令级并行（ILP）发挥的控制依赖，并避免分支误预测带来的性能损失。编译器不再生成条件跳转，而是生成一段单一的直线序列，其中包含带谓词的指令。这种变换对于实现向量化和软件流水线等其他优化至关重要，对于数字信号处理器等专用架构尤其重要。最新研究会尽早进行这种转换 [19]，以防止编译器执行损害性能的优化。在 ARM 等架构上，这种方法使用谓词来有条件地提交指令结果。硬件会执行这些指令，但只有在控制谓词为真时才允许提交其结果；否则，该指令将被取消。

然而，x86 缺乏对硬件谓词化的支持，因此会进行推测执行，而这依赖于两条路径都能安全执行。consequently，编译器不会对包含内存指令的分支路径应用 if 转换，从而留下大量潜在的可优化代码。其次，if 转换会进行朴素变换：它必须先完整执行两条路径，再进行选择。这种“全部执行，之后选择结果”的模式可能引入过高的指令开销，尤其是对于包含大量代码的分支；尽管消除了误预测，却往往会降低性能。为缓解这一问题，人们提出了各种方法，例如

使用 PGO 有选择地应用该变换 [33, 35]。其他混合技术（例如 Wish Branches [16], ）则将决策推迟到运行时，通过在二进制文件中同时编码分支及其 if 转换版本来实现。尽管采取了这些缓解措施，低效的完整推测模型在很大程度上仍被保留。相比之下，MERIT 不再局限于分支级决策，而是在 IR 指令级上运行，通过融合相似操作来减少冗余工作，而不是对两条路径都进行完整推测。

2.3 序列比对

MERIT 改编了 Smith-Waterman 算法 [29]，该算法最初为生物序列比对而开发，用于比对不同控制流路径上的结构相似指令序列。该算法使用动态规划，通过对匹配（相似指令）、不匹配（不同指令）和间隙（仅存在于一条路径中的指令）进行评分，找出最优局部比对，使 MERIT 能够发现融合机遇，即使指令顺序不同或中间穿插了其他操作（详见第 4 节）。

3 动机

在本节中，我们通过两个示例说明 MERIT 如何通过融合（并添加多余的）两条路径上的指令来消除分支。

3.1 MERIT 如何消除分支

MERIT 转换的核心思路是在分歧路径中插入多余（即虚设）的指令，使两条路径执行相同的操作，从而避免显式分支。以清单 1 中的 to_upper 函数为例，它会将给定字符串中所有小写字母转换为大写字母，字符串长度为 SIZE。在遍历字符串的每个字符时，for 循环中的 if 条件会反复执行。由于该分支高度依赖字符串中的数据，因此难以预测。在 x86 上对 to_upper 进行测试发现，该函数中约有 27% 的动态分支被误预测，导致指令每周期极低，仅为 0.48。

to_upper_branchless 函数在语义上与 to_upper 相同。不过，循环中的 if 条件判断被消除了，循环体变成了不含任何分支的直线代码（在 x86 汇编中，三元运算符会被转换为 cmov指令）。此版本的分支误预测几乎为零，与原始版本相比，速度提升了 32×。在此版本中，无论 cond 的值是什么（它表示原始版本中分支的执行结果），都会执行将常数值 diff 加到字符串中字符上的运算。这不会影响正确性，

[1]仅供说明之用。标准编译器优化发生在 MERIT 转换之后。

```
1void to_upper(char *str) { 2 for(int i=0;
i<SIZE; i++) { 3 if(str[i]>='a' & str[i]<='z')
4 str[i] =str[i] +'A' -'a'; 5 } 6 }
```

```
8void to_upper_branchless(char *str){ 9 for(int i
=0; i<SIZE; i++){ 10 bool cond = (str[i]>='a') & (str[i]<=
'z'); 11 unsigned int diff = cond ? 'A'-'a' : 0; 12 str[i
] +=diff; 13 } 14 }
```

代码清单 1. 示例动机

因为在分支未被执行时，变换只会向字符添加一个零。MERIT 会自动识别含分支代码，并将其变换为无分支版本。请注意，str[i] += diff 看起来会在每次迭代中无条件执行。如果 diff 为 0，且 str[i] 是只读的，则可能发生段错误。不过，正如我们稍后将在图 2c 中展示的那样，如果采取回退路径，MERIT 会在内部写入一个安全的虚拟内存位置，而不是写入可能导致故障的地址。

3.2 MERIT 与 if 转换的区别

图 1 展示了一种情况：两个分支执行的计算在结构上完全相同，区别仅在于操作数。在依赖条件移动指令（x86）的主流架构上，if 转换采用推测式的“计算所有结果，再选择结果”方法，完整计算 if 和 else 路径的结果，然后使用条件移动指令（cmov）选择正确的最终结果。尽管这种策略能有效消除分支预测错误带来的惩罚，但由于只需要一个结果，却对每项算术运算都执行两次，因此引入了冗余工作。

相反，MERIT 利用指令级对齐来识别分支之间的结构相似性。它不重复计算，而是合并操作数。这种“选择操作数，只计算一次”的方法首先使用选择指令，根据分支条件选择正确的常量，然后使用选定的操作数，将每个算术操作仅执行一次。MERIT 通过在指令级合并计算，而非仅仅在已完全计算的结果之间进行选择，大幅减少了总操作数，从而提升了性能。

4 详细设计

本节将介绍编译器变换 MERIT，旨在静态合并程序中发散的控制流路径

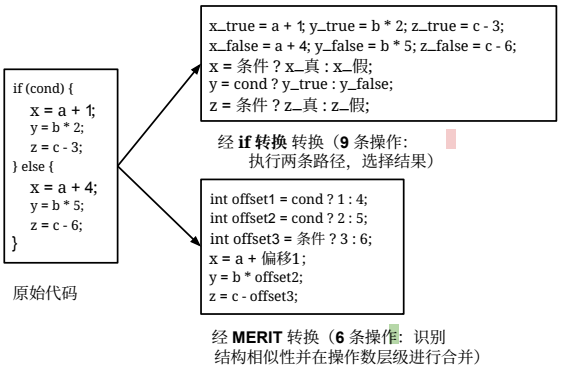


图 1。该示例表明，MERIT 可比 if 转换减少 33% 的运算：它识别出两条路径执行的是相同运算，只是常量不同，因此能够进行指令级合并，而非结果级选择。

。我们将说明插入冗余指令的阶段，使 if-then-else 语句中指令的操作序列保持一致。插入冗余指令有助于完全融合指令序列，从而消除代价高昂的分支（即难以预测的分支）。我们还将概述一份证明，说明 MERIT 是一种语义保持变换。

4.1 MERIT 转换

MERIT 转换基于与 DARM [24],相似的原理。DARM 是一种编译器优化，通过静态合并控制发散的程序路径来提升 GPU 程序的性能。DARM 主要识别具有相似基本块的 if-then-else 分支（即 同构控制流区域），并将这些基本块中的公共指令合并为收敛指令。DARM 的主要目标并非彻底消除分支；如果指令序列中的运算相同，分支便会随之消除。然而，两条分支路径上的指令序列包含相同运算的情况十分罕见。对于不相同的部分，DARM 会将其放入不同的基本块中以便有条件地执行，这会增加分支数量。

MERIT 的核心思想是在条件分支的两侧插入冗余指令，使两边的操作序列变得完全相同。当指令对齐中存在未对齐指令时，就需要冗余指令。我们可以通过以下定义，形式化地描述指令序列的对齐。考虑一个带有 if-then-else 分支的程序，该分支包含两个基本块 B_t 和 B_f（即菱形控制流），而 I_t 和 I_f 分别是这两个基本块的指令序列。

定义 1。兼容对：设(a, b) 是一对指令。当且仅当 a 和 b 的操作相同时，这一对指令才是兼容对（例如， a 和 b 的操作都是iadd）。指令 a 和 b 的操作数不必相同，(a, b) 也可以是兼容对。

定义 *2.* 指令对齐：设 $I_t = \{i_1^t < \cdots < i_n^t\}$ 和 $I_f = \{i_1^f < \cdots < i_m^f\}$ 分别为基本块 B_t 和 B_f 中的完整有序指令序列。指令对齐是由有序指令对组成的序列$A = \{(a_1, b_1) < \cdots < (a_k, b_k)\}$，满足 $a_j \in I_t \cup \{\psi\}, b_j \in I_f \cup \{\psi\}$。序列长度 k满足 $max(n, m) \leq k \leq n + m$，且 $\forall j \in [1, k], (a_j, b_j) \neq (\psi, \psi)$。如果 $a_j \neq \psi$和$b_j \neq \psi$，则 (a_j, b_j) 是可用于合并的兼容对。此处，ψ表示不存在指令（即空槽位）。

定义 3. 未对齐指令：设 $(a_i, b_i) \in A$是指令对齐 A 中的一对，且满足 $a_i = \psi$或 $b_i = \psi$。设 i'是指令对 (a_i, b_i) 中的有效指令，则 i'称为未对齐指令。

定义 4. 完整对齐：如果一条指令对齐 A' 不包含任何未对齐指令（即，对于所有指令对$(a_j, b_j) \in A', a_j \neq \psi$和 $b_j \neq \psi$），则称其为完整。这意味着对齐基本块中的指令序列长度必须相等（$n = m$）。

如果 B_t 和 B_f的指令对齐是完整的，我们就可以将 B_t 和 B_f完全合并为一个基本块，从而消除条件分支。MERIT 的第一步是将对齐 A 转换为完整对齐 A'。

定义 5. 冗余指令：设 I_t 包含一条未对齐指令 i'。将一条指令 i'' 插入 I_f，使得 i' 和 i'' 兼容，并在完整对齐 A' 中构成一对。i'' 被定义为冗余指令。尽管其功能与推测谓词化类似，*MERIT* 使用语义安全分析生成 i''，以确保该指令在原本不存在它的路径上执行时不会触发异常（例如除零、内存陷阱）。

冗余指令对于完整地对齐并消除分支至关重要。不过，必须确保该变换不会改变程序的语义。

选择指令最小化。在 MERIT 的代码生成过程中，如果两条合并指令的操作数不匹配，就会插入额外的选择操作。这一过程会增加程序中的指令数量，而额外的选择操作也会使数据流更加复杂。因此，最小化 MERIT 转换生成的选择指令数量至关重要。如果条件分支两侧具有相似的定义-使用链，就可以减少选择操作。更确切地说，设 $i_t = op(o_t^1, o_t^2)$ 和 $i_f = op(o_f^1, o_f^2)$ 是对齐 A 中的两条对齐指令。合并 i_t 和 i_f不

需要额外的选择操作，前提是 $o_t^1 = o_f^1$和 $o_t^2 = o_f^2$或 (o_t^1, o_f^1) 和 (o_t^2, o_f^2) 也由 A中的对齐指令生成。例如，考虑两条对齐指令i_t：r1 = t1 + 4 和 i_f：r2 = t2 + 4。如果 $t1$ 和 $t2$ 是先前已对齐并合并的指令对的结果（生成单个值 t_{new}），且常量操作数 4 相同，那么 MERIT 会生成一条合并指令 r_new = t_new + 4。无需额外的选择指令，因为这些操作数要么是共享的常量，要么直接来自已合并的数据依赖链。

IR 对齐评分。为了量化两条路径中定义-使用链的相似度，MERIT 引入了以下公式：
$Score = num_matches \times match_bonus - num_gaps \times gap_penalty$。本质上，评分表示匹配指令所带来的净收益（不引入额外的选择操作）。如果评分低于默认阈值（0.2，可在编译期间调整），MERIT 就会拒绝此次转换。

冗余指令的操作数。冗余 ALU 指令的操作数设置具有一定灵活性。我们可以根据指令的语义，将额外指令的所有操作数设为某个安全的常量值（例如，加法指令设为 0，除法指令设为 1）。另一方面，我们也可以设置操作数，以保留定义-使用链并尽可能减少选择指令。在 MERIT 中，我们结合采用这两种方式。如果额外指令不会导致失败（例如溢出、下溢、除零或未定义行为），我们就保留定义-使用链并尽可能减少选择操作。本节末尾将通过一个示例说明操作数设置过程。

对齐内存
操作。
• 如果我们能在编译时确定两个对齐的内存操作访问的是同一地址，就会将它们合并为一个内存操作（无需使用选择指令）。

• 如果两个对齐的内存操作访问不同的内存位置，我们会将它们合并，并根据条件为合并指令选择地址。

• 如果存在未对齐的内存操作，我们会在对齐的空槽中插入一条冗余内存指令，以访问由 MERIT 定义并经过正确初始化的安全全局内存空间中的某个位置。该安全全局内存空间由编译器分配并初始化，且保证不会与程序中的任何其他内存位置发生别名。

• 为确保线程安全，编译器会为每个线程准备独立的内存空间。

示例。 现在，我们使用贯穿全文的示例（代码清单 1）说明 MERIT 转换的实际工作方式。图2展示了 to_upper 函数在各个阶段的转换过程。图 2a 展示了 to_upper 函数，其中插入了一个空的 else 部分。这是 MERIT 的一个额外规范化步骤，它将 if-then 转换为 if-then-else形式，从而使其能够合并 if-then 分支。此外，为了提高可读性，指令按行分别显示（第 4–6 行）。图 2b 展示了插入冗余代码后的代码。此处，else 路径为空，因此所有指令都未对齐。else 路径中包含插入的冗余指令。例如，冗余代码插入后，第 4 行中的加载操作会在第 9 行重复执行，并使用来自安全全局内存空间的位置（mem）。MERIT 还会尝试保留定义-使用链，并尽量减少合并所需的选择操作。例如，第 11 行的变量 t7 使用第 10 行的 t6。这类似于变量 t3 将 t2 用作第一个操作数。t7 将 0 用作第二个操作数，以避免任何溢出/下溢错误。此示例还展示了冗余代码插入期间如何处理存储指令。在 if 路径中，会将值 t3 存储到 text[i]（第 7 行）。在 else 路径中，也会执行相同的存储操作（第 12 行），但存储位置为 mem，存储值为 t7。图 2c 展示了合并步骤 [2]后的代码。请注意，插入了额外的选择指令（以三元运算符的形式显示），用于在输入操作数不匹配时选择操作数。转换后的程序执行速度比原始程序快得多（第 3 节）。

汇总。我们在算法 1 中描述了针对程序的整体 MERIT 转换。MERIT 转换会遍历程序中的所有函数。对于每个函数，它会收集所有适用于 MERIT 转换的有效分支。分支的结构有效性 由分支的两条路径在某个基本块处汇聚并形成直线控制流来确定。换言之，即 if-then-else（即，双侧分支）或 if-then（即，单侧分支）的控制流区域。然后，对于每个有效分支，它会计算分支两侧的指令对齐情况。如果无法完成对齐，则会插入冗余指令以完成对齐。对于单侧分支（if-then）这一退化情况，将添加一个完整的由冗余指令组成的基本块。最后，它会合并两个基本块并简化函数，从而移除分支。我们会重复执行这些步骤，直到函数不再发生变化。并非所有结构有效的分支都需要进行 MERIT 转换。决定哪些有效分支需要转换是灵活的，最好通过配置文件引导优化（PGO）来选择性地针对具有挑战性且难以预测的分支（第 5.2 节）。

MERIT 的正确性***MERIT***。MERIT 转换是一种语义保持变换，不会导致程序崩溃。不失一般性，

[2]用于逐步说明融合过程的详细序列。标准优化遍（常量传播、死代码消除（DSE））会在 MERIT 之后运行，以清理常量。

```
// ...
如果 ((文本[i] >= 'a')
  & (文本[i] <= 'z')) {
  t1 = 文本[i] -  'a';
  t2 = t1 +  'A';
  文本[i] =t2;
} 否则 {
}
```

(a) 转换为_大写函数，带有一个插入了空的 else 代码段。

```
// ...
如果 ((文本[i] >= 'a')
  & (文本[i] <=  'z')) {
  t1 = 文本[i];
  t2 = t1 -  'a';
  t3 = t2 + 'A';
  文本[i] = t3;
} 否则 {
  t5 = *mem;
  t6 = t5 - 0;
  t7 = t6 + 0;
  *mem = t7;
}
```

(b)代码位于多余代码之后插入。

```
// ...
无符号 是_小于 =
  (文本[i] >=  'a') & (text[i] <= 'z');
t1_t5 = 是_小于 == 0 ? 文本[i] : *mem;
s1 =is_小于 == 0 ? 0 :    'a';
t2_t6 = t1_t5 - s1;
s2 =is_小于 == 0 ? 0 :    'A';
t3_t7 = t2_t6 + s2;
t4_t8 = is_小于 == 0 ? &text[i] : mem;
*t4_t8 = t3_t7;
```

(c) 合并步骤后的代码。

图2. MERIT 转换示例

算法 1： MERIT转换算法

输入： 原始程序 P 和排除位置 L
输出： 转换后的程序 P'
对于 $F \in$ 个函数*(P)* **执行**
  已更改 ← **false**;
  重复
    有效分支 ← collectValidBranches(F, L);
    for *BI* ∈ 有效分支 **do**
      BL、BR ←获取菱形块(BI);
      A ← 计算对齐(BL, BR);
      如果 判断能否完成对齐*(A, BL, BR)* **为假，则**
        继续
      end
      BL'、BR' ← 插入额外指令(A, BL, BR);
      合并代码块(BL'、BR');
      Changed ← 真;
      如果*Changed* **为真，则**
        simplify(F);
      end
    end
  直到*Changed* **为假**;
end

我们可以推理，将 MERIT 应用于单个分支不会影响正确性。如果分支两侧的指令序列在操作上相同，那么用选择指令融合这些序列是语义保持的。只需证明插入冗余指令不会影响程序的正确性。我们可以证明，任何冗余指令都不会影响程序原有的数据流，因为冗余指令产生的值永远不会被由程序的原始指令使用。此外，冗余指令绝不能写入程序的内存位置。因此，即使插入冗余指令，程序的数据流仍保持不变。此外，冗余指令不会访问在插入该指令的路径中不应访问的内存位置。因此，MERIT 转换不会影响程序的正确性，也不会导致崩溃。

5 实现

5.1 编译器集成

我们在 LLVM-14 中将 MERIT 实现为一个 LLVM pass，集成到标准流水线中，并默认在高于 -O0 的优化级别启用。它被策略性地安排在早期规范化和简化编译遍（例如，SimplifyCFG、SROA 和 Mem2Reg）之后。这样的安排是有意为之：这使 MERIT 能够基于 LLVM 现有的控制流合并优化开展工作。此外，Mem2Reg 等早期简化编译遍会移除冗余代码，从而缩小并简化中间表示。这降低了 MERIT 指令对齐步骤的编译时复杂度。由于在后端优化之前运行，MERIT 还可避免与后续阶段的别名分析或内存布局变更发生潜在冲突。

5.2 启用配置文件引导优化

虽然 MERIT 能通过针对控制流模式进行指令级合并来实现高性能，但在程序的所有分支中不加区分地应用这种变换，可能会导致性能下降。在分支

预测准确率已经很高，或合并后的指令序列明显长于原始分支代码时，选择指令和操作数多路复用所引入的开销可能超过分支消除带来的收益。我们通过以配置文件引导优化（PGO）的方式改进 MERIT 的应用来应对这一挑战：根据观测到的实际性能，有选择地进行变换。

MERIT 通过编译器命令行选项提供了细粒度的筛选机制，可按源文件名、函数名或行号进行筛选，从而支持有选择地应用变换。例如，开发者可以使用 -include-func-names=func1,func2 指定函数级包含或排除列表，仅对高价值目标启用 MERIT（例如，热点函数或不可预测分支数量最多的函数），也可以使用 -exclude-func-names=func3,func4 将有问题的函数列入黑名单（例如，已知会产生较高开销的函数）。此外，MERIT 支持通过 -exclude-file-names=utils.c,legacy.c 进行文件级筛选，以排除整个源文件；还支持通过 -json-include-lines=profile.json 实现行级精确控制，其中 JSON 文件将源文件名映射到应执行变换的行号数组。这种多粒度筛选基础设施使开发者能够通过排除性能特征较差的变换位置，同时保留表现出明显收益的位置，迭代完善其优化策略。

6 评估

6.1 实验设置

硬件和操作系统。所有测试均在一台 x86 服务器上运行，该服务器配备英特尔至强 Silver 4314 处理器、32 个核心和 192 GB 内存。其微架构为英特尔 Sunny Cove（Ice Lake），并采用基于 TAGE 的分支预测器。该系统运行 Ubuntu 22.04.5 LTS，基于内核版本 6.8.0。

工作负载。我们在四个基准测试套件中的共 102 个基准测试上评估 MERIT：

- 24分支密集型微基准测试，涵盖多种常见算法。
- 16 CPU SPEC2017 [6] 基准测试，采用速率模式编译。我们有意排除所有 Fortran 工作负载，因为 LLVM-14 缺少将 Fortran 降低为 LLVM IR 所需的 flang-new 代码生成功能。
- 40 由 MERIT 优化的 Python 解释器运行的 [31] pyperformance 基准测试。
- 22 由 MERIT 优化的 SQLite [9] 引擎处理的 TPC-H [32] OLAP 查询。

比较对象。我们将基准测试编译为四种模式进行评估：

- **MERIT**：应用该变换，并禁用后端优化。
- **MERIT-PGO**：根据启发式结果应用配置文件引导优化，以排除那些性能下降的函数（第 5.2 节）。
- **MERIT-O2**：应用该变换，并启用后端优化。
- **条件转换**：应用早期条件转换 [19]，并启用后端优化。

所有模式报告的性能均相对于各自对应的基线进行归一化（纯硬件预测，不进行任何特殊编译器变换）。为确保比较公平，该基线始终采用与所评估模式相同的后端优化级别进行编译。具体而言，MERIT 和 MERIT-PGO 与禁用后端优化的基线进行比较，而 MERIT-O2 和条件转换则与启用后端优化的基线进行比较。每项报告数值均取 5 次独立运行结果的中位数，运行间的差异可以忽略不计。

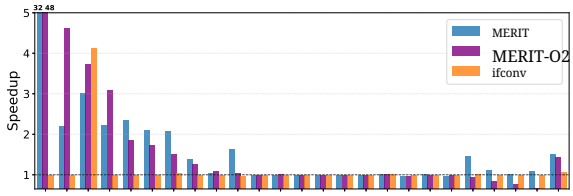
6.2 微基准测试

后端优化对微基准测试的影响。在图 3a 中，我们测量了 24 个分支密集型微基准测试的运行时间性能，涵盖排序、比较、遍历等任务。与硬件推测执行相比，MERIT 在 toUpper 中最高可实现 32× 的加速，在所有基准测试中，几何平均值（geomean）加速比为 1.51 倍。MERIT-O2 在 toUpper、ccomp、dutchFlag 和 bubbleSort 中的性能优于 MERIT，但在其他测试中表现较差（几何平均值为 1.44 倍）。请注意，在 arrayMerge、heapSort、A* 和 prim 中，MERIT-O2 会导致性能下降，而 MERIT 不会。例如，prim 在 MERIT 下性能提升了 8%。然而，启用 MERIT-O2 的后端优化后，性能反而下降了 33%。prim 的计算内核包含控制流，用于检查三个条件是否都满足。不巧的是，下游优化器将 MERIT 转换后的代码视为未经优化的选择指令，并将其还原为分支，每个条件检查对应一个分支。因此，与只有一个跳转目标的原始分支代码相比，MERIT-O2 后端生成了三个跳转目标，导致不必要的硬件误预测增加。

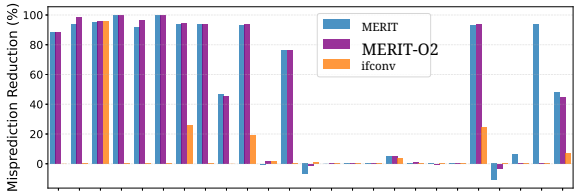
> 要点：由于 MERIT 在 LLVM 中间表示的早期阶段运行，现有的下游编译遍可能会以次优方式转换代码，包括将无分支代码还原为含分支代码。

早期条件转换带来的性能提升要小得多（几何平均值仅为 1.06 倍）。这是因为它无法转换存在不安全内存操作的控制流区域。例如，toUpper 在分支中包含 str[i]（代码清单 1），这可能会在 x86 上导致无效内存访问。而 MERIT 在融合 IR 指令时会安全地保护内存访问，从而避免了此类问题。

我们在 dutchFlag 中发现了一个罕见情况：MERIT 和 if 转换 都能带来改进，但 if 转换 优于 MERIT。该算法循环包含一个高度不可预测的三路分支，仅用于控制简单的计数器递增。早期条件转换在此处表现出色，因为它能将该分支转换为计算开销极低的 cmov指令，从而消除误预测惩罚。



(a) MERIT 的性能几何平均值达到 1.51，而 if 转换为 1.06。

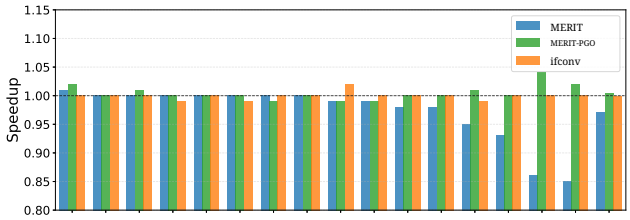


(b) MERIT 平均实现了 48% 的分支未命中率降低，而 if 转换为 7.2%。

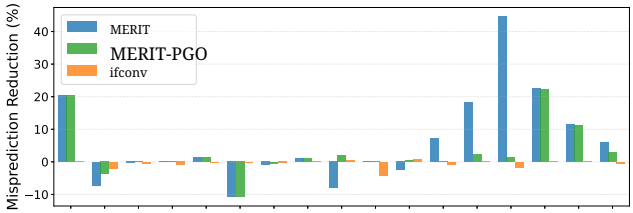
图3. 与硬件推测（不进行变换）相比，MERIT 和早期条件转换在**微基准测试**上的性能与分支未命中率降低幅度。

这种特定工作负载非常适合 if 转换，因为它不包含 MERIT 旨在处理的复杂且不安全的内存操作。此外，作为较后阶段的编译遍，极低的寄存器压力和简单的数据依赖链意味着后端优化（寄存器分配、指令调度）能够很好地处理早期条件转换的输出；相比之下，MERIT 的 IR级选择操作可能会经历额外的降级变换，从而引入相对低效。

CPU SPEC2017 性能。 图 4a 展示了经 MERIT 编译遍变换后的 SPECrate 2017 性能。遗憾的是，在大多数基准测试中盲目应用 MERIT 会导致性能下降（蓝色柱）。这是因为大多数基准测试的分支并不密集，因此 MERIT 会产生过多的运行时指令，与不进行变换相比，几何平均值为 0.97。不过，当我们应用配置文件引导优化（PGO）来识别并排除导致性能下降最严重的函数后，所有基准测试的性能都持平或有所提升（绿色柱）。以 505.mcf_r 为例，朴素地应用 MERIT 会导致性能下降 14%，但识别并排除那些表现不佳的函数后，运行时间可提升 4%。if 转换与我们在微基准测试中看到的情况相同，几乎没有表现出任何性能



(a) MERIT 的性能几何平均值为 0.97，MERIT-PGO 为 1.01，而 if 转换为 1。



(b) 与 if 转换相比，MERIT 平均实现了 6.1% 的分支未命中率降幅，MERIT-PGO 则实现了 3% 的降幅；if 转换的降幅为 -0.6%。

图4. MERIT 和早期条件转换在**SPECrate 2017**上的性能与分支未命中率降低幅度。

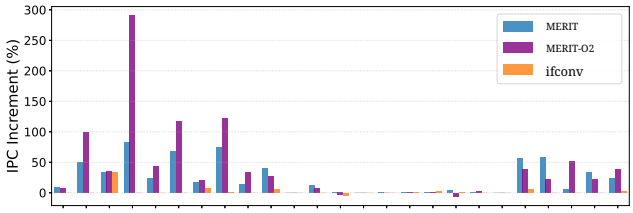
差异表明，在生成无分支代码时，它较为保守。

误预测减少。 我们评估 MERIT 相较于条件转换消除分支误预测的效果。在微基准测试中（图 3b），MERIT 平均减少 48% 的分支误预测；在 SPECrate 2017 中（图4b），平均减少 6.1%。与早期条件转换相比，MERIT 在我们的微基准测试中减少的误预测多 40.8%，在 SPECrate 2017 中则多 6.7%。对于微基准测试，误预测减少的整体趋势与图 3a 中的性能提升相符。然而，对于更贴近实际的 SPECrate 基准测试，误预测减少并不直接对应性能提升。例如，在 505.mcf_r 中，尽管 MERIT 和 MERIT-PGO 减少的误预测数量相同，MERIT-PGO 的性能提升了 4%，而 MERIT 的性能却下降了 14%。在 538.imagick_r 中，尽管 MERIT 减少了 44% 的误预测，性能却反常地下降了 7%。

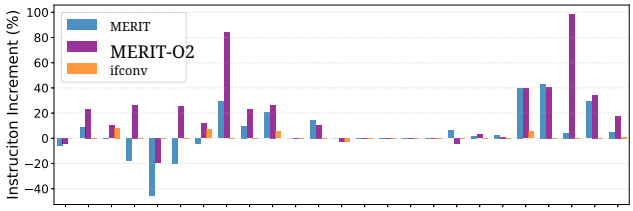
要点：对于复杂工作负载，仅仅专注于减少误预测并不一定能提升性能。采用智能的成本模型，避免应用会带来沉重开销的优化，是有益的。

IPC 性能提升。 每周期指令数（IPC）直接反映了编译器如何转换程序，以利用处理器的指令级并行。如图 5a 所示，MERIT 和 MERIT-O2 显著提升了微基准测试的 IPC。然而，对于复杂的 SPECrate 工作负载（图 6a），IPC 与性能之间的关系则更加微妙。例如，在 508.namd_r 和 505.mcf_r 中，尽管 IPC 分别下降了 20% 和 26%（图 6a），MERIT-PGO 仍分别实现了 2% 和 4% 的加速（图 4a ）。

这种看似矛盾的现象，可以用退役指令总数来解释。基线（未进行变换）会频繁发生误预测，迫使 CPU 推测执行并退役大量沿错误路径执行的指令。MERIT 完全消除了这些误预测路径。虽然 MERIT 会在正确路径上增加少量冗余指令，但增量远小于被消除的错误路径指令数量。因此，退役的动态指令总数（I nstructions_new）显著减少。总周期数（Cycles_new）和总指令数（Instructions_new）都减少了，但指令数减少得更多，因此 IPC（Instructions_new / Cycles_new）降低，即使总执行时间缩短了（实现了加速）。要点：MERIT 插入冗余指令只是其中一部分。它的主要优势在于消除了误预测路径上数量大得多的指令开销。在复杂工作负载中，这可能使动态指令总数净减少，从而导致 IPC 下降，即使实际运行时间有所改善。



(a) MERIT 平均实现了 24.4% 的指令每周期性能提升，而 if 转换的平均提升为 2.1%。



(b) MERIT 平均产生了 17.8% 的运行时指令开销，而 if 转换的平均开销为 0.9%。

图 5。 MERIT 和早期条件转换在微基准测试中相对于 no 转换的指令每周期增幅和运行时指令开销。

动态指令开销。 分支消除的关键考量是动态指令数量的增长。按照设计，我们的变换会将控制流转换为数据流，这通常会导致原始分支两条路径上的指令都被执行。图 5b 和图 6b 分别量化了微基准测试和 SPECrate 套件中的这一开销。

在我们的微基准测试中，许多工作负载的动态指令数量增加了 10% 到 100%。A* 基准测试清楚地展示了这种开销如何导致性能下降。然而，这种开销并不总是与性能损失相关。像 qsort、arrayMerge 和

heapSort 均获得了性能提升，尽管存在运行时指令开销。这意味着，与频繁且严重的分支误预测所造成的惩罚相比，这些额外指令的成本可以忽略不计。

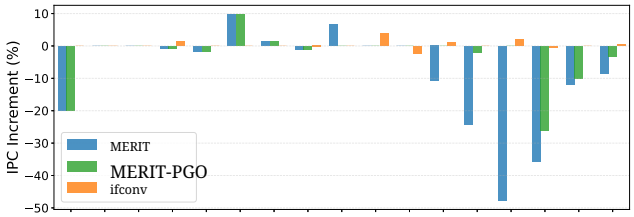
在 SPECrate 套件（图 6b）上，朴素 MERIT 的平均开销为 2.7%。这凸显了 MERIT-PGO 的重要性：它通过在指令开销无法被消除误预测所带来的收益抵消的函数上选择性地禁用变换，将该开销降至仅 0.6%。

6.3 案例研究：SQLite

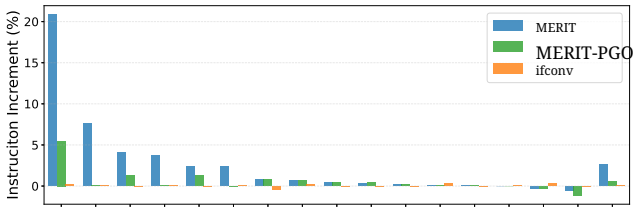
我们的第一个案例研究展示了基于剖析的选择性转换的必要性：一台由 MERIT 编译、运行 22 个 TPC-H 查询的 SQLite 引擎清楚地证明了这一点。

如图 7 所示，直接应用 MERIT（蓝色柱状图）会导致许多查询的性能显著下降，尤其是查询 7、8、9 和 21，几何平均值降至 0.84。这同样是因为转换那些并非性能瓶颈的分支带来了指令开销。

然而，MERIT-PGO（绿色柱形）的结果带来了显著变化。通过使用配置文件引导优化识别并排除问题函数，我们完全消除了所有性能减速，并实现了 1.01 的正向几何平均性能。此外，MERIT-PGO 通过仅针对高价值、易发生误预测的分支，在若干查询（例如 5、6、11、15 和 18）上实现了加速。这种选择性变换印证了我们的结论：对于复杂的真实应用，MERIT 必须搭配智能成本模型或配置文件引导优化筛选，才能提升性能。条件转换不会带来任何性能差异，这可能是因为它无法变换性能关键代码块。



(a) 与条件转换带来的 0.3% IPC 提升相比，MERIT 的 IPC 性能下降 8.5%，MERIT-PGO 的 IPC 性能下降 3.2%。



(b) 与条件转换平均 0.1% 的开销相比，MERIT 的运行时指令开销为 2.7%，MERIT-PGO 的开销为 0.6%。

图 6. MERIT 与早期条件转换在**SPECrate2017**上的 IPC 增量和运行时指令开销。

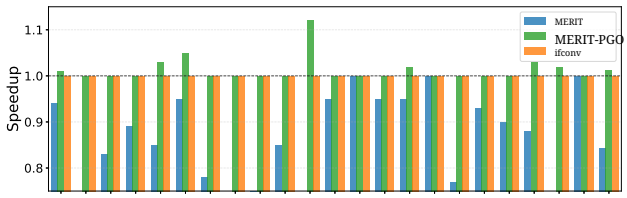


图 7. MERIT 的性能几何平均值为 0.84，MERIT-PGO 为 1.01，而条件转换为 1。

6.4 案例研究：Python

在第二个案例研究中，我们评估了 MERIT 在一个大型真实应用中的表现：CPython解释器，其中包含来自 pyperformance套件 [31]的 40 个基准测试。如图 8 所示，不加区分地应用 MERIT 可实现 1.01 的几何平均加速；一些工作负载（如 unpickle_list 和 nbody）受益，而其他工作负载则没有。相比之下，标准的 if-conv优化通道波动更大，表现为

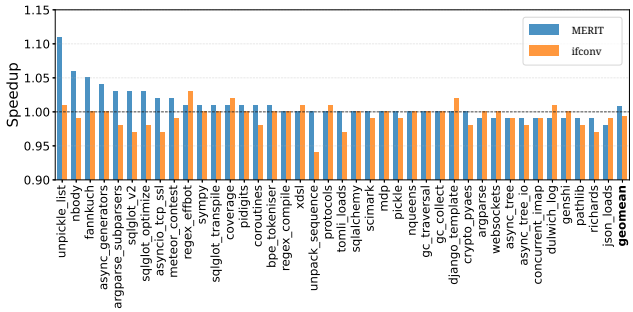


图8. 与条件转换的 0.99 相比，MERIT 的性能几何平均值为 1.01。

在某些工作负载上有所提升（如 regex_effbot），但在另一些工作负载上则显著下降（unpack_sequence），几何平均值为 0.99。

我们没有报告 Python 套件上的 MERIT-PGO 性能，因为对于解释器这类复杂运行时环境而言，其函数级过滤粒度过粗；pyperformance 套件中的不同工作负载会对代码的不同部分造成压力。排除某个函数以优化一种工作负载（例如，nbody），可能会损害另一种依赖同一函数变换的工作负载（例如，json_loads）。不过，这也为专用环境提供了一种实用策略：可以使用 MERIT 和配置文件引导优化，创建一个高度调优、仅针对特定工作负载优化的解释器（例如 JSON序列化服务），只变换其关键路径中热点且易误预测的函数，从而实现那些在通用基准测试中会被抵消的加速。

7 局限性与未来工作

我们确定了限制 MERIT 有效性的两个关键局限：缺乏用于选择性变换的静态成本模型，以及 MERIT 与后端优化通道之间缺乏协调。

对成本模型的需求。 当前 MERIT 的适用范围有限：只要 IR 对齐分数达到阈值，它就会对任何符合条件的分支统一应用该变换。这种粗粒度的应用可能适得其反，尤其是在分支本身已经具有很高的可预测性（例如循环不变）或合并控制流路径会引入过多指令开销时。由于 MERIT 在目标无关 IR 级别运行，缺乏后期 Machine-IR优化通道所具备的指令调度和延迟信息，因此开发用于估算收益的细粒度成本模型是一项并不简单的挑战。

未来的工作是提出一种基于启发式的 MERIT变换成本模型，统一考虑 IR对齐分数、由此产生的静态指令开销，以及新 select链的数据流依赖深度。这可能需要在

早期 LLVM 中间表示阶段积极应用 MERIT，并在后期机器中间表示阶段部分还原为含分支代码，具体取决于仅在编译后期才能获得的资源使用情况 [4]。

然而，一个根本性挑战仍然存在：编译器必须知道硬件的分支预测器是否会在某些难以预测的分支上失效。这需要比当前简单的配置文件引导优化更细粒度的配置文件引导优化，以识别每个分支的分支预测错误开销。如果配置文件引导优化不适用，另一个有前景的方向是探索利用大型语言模型（LLM）或基于 Transformer 的执行估计模型（如 TRACED [8] ）来构建强大的人工智能驱动的代价模型。该模型可以在海量代码库上进行微调，从而识别简单启发式方法可能遗漏的复杂模式。这种方法旨在使 MERIT 更接近零成本且高效的静态优化。

缺乏*MERIT*感知的后端优化。第二个限制是 MERIT 与后端优化通道之间缺乏协调，且目标存在冲突。下游处理流程可能会撤销 MERIT 的工作，或作出次优决策。我们观察到了一种“性能异常”：激进的后端优化有时会使原始分支代码获得更大的相对收益，从而掩盖 MERIT 的真正潜力。

这可能由多种原因导致：后续编译遍可能会预期代码中含有分支，以便应用寄存器压力降低或目标特定谓词化等优化。这可能会撤销 MERIT 的变换。未来的工作将通过使用 LLVM 元数据，将编译器提示从 MERIT 传递给后端编译遍，以保留无分支结构，从而解决这一问题。

8 相关工作

已有多种技术被提出用于改进分支预测，包括利用取指引导的指令预取（FDIP）[10, 23], 、类 TAGE [25–28], 以及基于感知机的方法 [12, 13]。尽管这些技术旨在实现高准确率，但它们难以应对嘈杂的历史记录，并且常常受数据中心应用庞大的分支足迹影响，导致频繁出现容量引起的缺失。

混合方法利用软件配置文件引导优化（PGO）。这项技术广泛应用于数据中心：在生产环境中收集应用程序配置文件，并用于离线分析 [5, 7, 14, 20,21]。Whisper [15] 分析离线配置文件，生成轻量级布尔公式，并将其作为预测提示注入二进制文件。BranchNet [34] 利用离线分析训练复杂的机器学习模型，再通过小型片上推理引擎将模型提供给运行时，以预测最难处理的分支。这些技术利用离线计算资源解决硬件在运行时无法学习的预测问题，从而增强硬件预测器。

9 结论

本文介绍了 MERIT，这是一种新颖的编译器变换，通过融合结构相似的操作序列来消除 x86 上的分支。与传统的 if 转换不同，MERIT 的“选择操作数，计算一次”策略利用语义分析实现安全的操作数级保护，因此能够安全地变换包含条件内存操作的分支，而其他方法必须跳过这类分支。我们还展示了如何通过支持选择性变换，将 MERIT 轻松集成到配置文件引导优化（PGO）中。在 102 个工作负载上，MERIT 平均实现了 10.9% 的加速（最高达 32×）。我们的分析发现，MERIT 的效果目前受限于缺乏静态代价模型，以及可能撤销其成果的“对 MERIT 视而不见”的后端优化；这也凸显了它们是未来研究的关键方向。

参考文献

[1] KA Jordan Alexander 和 Kim Nikolai。 [n. d.]。面向谓词架构的 IR 级与机器级 if 转换。载于*DSP* 与嵌入式系统优化第*10*届研讨会论文集。3–10。[2] John R Allen、Ken Kennedy、Carrie Porterfield 和 Joe Warren。1983。将控制依赖转换为数据依赖。载于第*10*届 *ACM SIGACT-SIGPLAN* 编程语言原理研讨会论文集。177–189。[3] David I August、Wen-mei W Hwu 和 Scott A Mahlke。1997。平衡控制流与谓词化的框架。载于第*30*届年度国际微体系结构研讨会论文集。IEEE，92–103。[4] David I August、Wen-Mei W Hwu 和 Scott A Mahlke。1999。用于平衡控制流与谓词化的部分逆向 if 转换框架。《国际并行编程期刊》27，5（1999），381–423。[5] Grant Ayers、Nayana Prasad Nagendra、David I August、Hyoun Kyu Cho、Svilen Kanev、Christos Kozyrakis、Trivikram Krishnamurthy、Heiner Litz、Tipp Moseley 和 Parthasarathy Ranganathan。2019。Asmdb：理解并缓解仓库级计算机中的前端停顿。载于第*46*届国际计算机体系结构研讨会论文集。462–473。[6] James Bucek、Klaus-Dieter Lange 和 Jóakim v. Kistowski。2018。SPEC CPU2017：新一代计算基准测试。载于*2018*年*ACM/SPEC*国际性能工程会议伴随论文集。41–42。[7] Dehao Chen、David Xinliang Li 和 Tipp Moseley。2016。AutoFDO：面向仓库级应用的自动反馈引导优化。载于*2016*年国际代码生成与优化研讨会论文集。12–23。[8] Yangruibo Ding、Benjamin Steenhoek、Kexin Pei、Gail Kaiser、Wei Le 和 Baishakhi Ray。2024。TRACED：面向源代码的执行感知预训练。载于第*46*届*IEEE/ACM*国际软件工程会议论文集。1–12。[9] Kevin P Gaffney、Martin Prammer、Larry Brasfield、D Richard Hipp、Dan Kennedy 和 Jignesh M Patel。2022。SQLite：过去、现在与未来。《*VLDB* 捐赠会刊》15，12（2022）。[10] Yasuo Ishii、Jaekyu Lee、Krishnendra Nathella 和 Dam Sunwoo。2020。重新定位指令预取：产业视角。《*IEEE* 计算机体系结构快报》19，2（2020），147–150。[11] Daniel A Jiménez。 [n. d.]。多视角感知机预测器与 TAGE。（[n. d.]）。

[12] Daniel A Jiménez。2016。多视角感知机预测器。载于第*5*届 *JILP* 计算机体系结构竞赛研讨会（*JWAC-5*）：锦标赛分支预测（*CBP-5*）。[13] Daniel A. Jiménez 和 Calvin Lin。2001。基于感知机的动态分支预测。载于第7届国际高性能计算机体系结构研讨会（*HPCA*）论文集。IEEE，197–206。https://doi.org/10.1109/HPCA.2001.903263[14] Tanvir Ahmed Khan、Nathan Brown、Akshitha Sriraman、Niranjan K Soundararajan、Rakesh Kumar、Joseph Devietti、Sreenivas Subramoney、Gilles A Pokam、Heiner Litz 和 Baris Kasikci。2021。Twig：面向数据中心应用的基于配置文件引导的 BTB 预取。载于*MICRO-54*：第*54*届*IEEE/ACM*国际微体系结构研讨会。816–829。[15] Tanvir Ahmed Khan、Muhammed Ugur、Krishnendra Nathella、Dam Sunwoo、Heiner Litz、Daniel A Jiménez 和 Baris Kasikci。2022。Whisper：面向数据中心应用的基于配置文件引导的分支预测错误消除。载于*2022*年第*55*届*IEEE/ACM*国际微体系结构研讨会（*MICRO*）。IEEE，19–34。[16] Hyesoon Kim、Onur Mutlu、Yale N Patt 和 Jared Stark。2006。Wish Branches：实现自适应且积极的谓词执行。*IEEE Micro* 26，1（2006），48–58。[17] Chit-Kwan Lin 和 Stephen J. Tarsa。2019。分支预测并非已解决的问题：测量、机遇与未来方向。arXiv: 1906 08170 [ ] 1906 08170 。cs.AR [18] 2018 https://arxiv.org/abs/ 。Pierre Michaud。 。一种替代性的 TAGE 类条件分支预测器。《*ACM*体系结构与代码优化汇刊》（*TACO*）15，3（2018），1–23。[19] Dorit Nuzman、Ayal Zaks 和 Ziv Ben-Zion。2024。尽可能早地进行If-Convert。载于第*33*届*ACM SIGPLAN*国际编译器构造会议论文集（爱丁堡，英国）（*CC 2024*）。计算机协会，纽约州纽约市， 26 38 10 1145 3640537 3641562 – 。 [20] https://doi.org/ 。 / 。Maksim Panchenko、Rafael Auler、Bill Nell 和 Guilherme Ottoni。2019。Bolt：适用于数据中心及其他场景的实用二进制优化器。载于*2019*年*IEEE/ACM*代码生成与优化国际研讨会（*CGO*）。IEEE，2–14。[21] Maksim Panchenko、Rafael Auler、Laith Sakka 和 Guilherme Ottoni。2021。Lightning bolt：强大、快速且可扩展的二进制优化。载于第*30*届*ACM SIGPLAN*国际编译器构造会议论文集。119–130。[22] Joseph CH Park 和 Mike Schlansker。1991。论谓词执行。惠普实验室，加利福尼亚州帕洛阿尔托。[23] Glenn Reinman、Brad Calder 和 Todd Austin。1999。取指引导的指令预取。载于*MICRO-32*：第*32*届*ACM/IEEE*年度国际微体系结构研讨会论文集。IEEE，16–27。[24] Charitha Saumya、Kirshanthan Sundararajah 和 Milind Kulkarni。2022。DARM：用于减少 SIMT 线程分歧的控制流融合——扩展版。arXiv:2107.05681 [cs.PL] https://arxiv. org/abs/ 2107 05681 。André Seznec。 。一个64千字节的 ISL-TAGE分支预测器。载于*JWAC-2*：锦标赛分支预测。[25] 2011 [26] André Seznec。2014。Tage-sc-l分支预测器。载于*JILP*锦标赛分支预测。[27] André Seznec。2016。再次研究Tage-sc-l分支预测器。载于第*5*届 *JILP*计算机体系结构竞赛研讨会（*JWAC-5*）：锦标赛分支预测（*CBP-5*）。[28] André Seznec 和 Pierre Michaud。1999。去别名混合分支预测器。博士学位论文。法国国家信息与自动化研究所。[29] Temple F Smith、Michael S Waterman 等。1981。共同分子子序列的识别。《分子生物学杂志》147，1（1981），195–197。[30] Shixin Song、Tanvir Ahmed Khan、Sara Mahdizadeh Shahri、Akshitha Sriraman、Niranjan K Soundararajan、Sreenivas Subramoney、Daniel A Jiménez、Heiner Litz 和 Baris Kasikci。2022。Thermometer：面向数据中心应用的配置文件引导 BTB 替换。载于第*49*届国际计算机体系结构年会论文集。742–756。[31] Victor Stinner。 [无日期]。Python 性能基准测试套件 — Python 性能基准测试套件 1.0.6 文档。https: //pyperformance.readthedocs.io/ [32] 事务处理性能委员会（TPC）。2023。*TPC* 基准测试 *H*（决策支持）标准规范。技术报告，修订版 3.0.1。事务处理性能委员会。https://www.tpc.org/tpch/ [33] Li Wang、Hong An、Yongqing Ren 和 Yaobin Wang。2008。面向数据流谓词化的配置文件引导优化。载于*2008* 年第*13*届亚太计算机系统结构会议。IEEE，1–8。[34] Siavash Zangeneh、Lizy K. John 和 Andreas Gerstlauer。2020。Branch-Net：一种用于预测难以预测的分支的卷积神经网络。载于第*53*届 *IEEE/ACM* 国际微体系结构研讨会（*MICRO*）论文集。IEEE，118–130。https: //doi.org/ 10 1109 50266 2020 00020 。/MICRO。。Eric J [35] 2005 Zimmerman。。超标量微处理器中的配置文件引导 *if* 转换。博士学位论文。伊利诺伊大学厄巴纳-香槟分校。