编译器优化技术及指令调度算法的研究

作者：陈朴炎

学号：2021211138

报告时间：2024.6.2

摘要：本研究报告主要探讨了两项编译器优化技术：基于有向无环图的指令调度算法和基于图着色的寄存器分配算法。首先，通过构建 DAG 图来捕获汇编指令间的依赖关系，并设计了一种调度策略，该策略通过排序和选择根顶点来最大化指令的并行执行，从而提高了处理器的利用率。接着，研究了基于图着色的寄存器分配算法，通过为程序中的变量合理分配寄存器以减少内存访问次数，进而提升了程序的执行效率。这两项技术的深入研究和实现不仅为编译器优化提供了新的视角，也为提高程序执行性能提供了有效手段。

1 指令调度算法

1.1 相关概念

1.1.1 基本块的 DAG 图的定义

在编译过程中，DAG 图[i]用于表示基本块内指令间的依赖关系。

G = (V, A)：这里 G 表示 DAG 图，V 表示图中的顶点集合，每个顶点代表一个指令，A 表示图中的弧集合，弧表示指令之间的依赖关系。

弧的权值 w(v1, v2)：表示指令 v2 必须至少在指令 v1 执行完 w(v1, v2) 个周期后才能被执行。

顶点的直接前驱集合 prev(v)：对于顶点 v，直接前驱集合是指所有能够直接影响 v 执行的指令集合。即 {n | n ∈ V 且 (n, v) ∈ A}。

顶点的直接后继集合 succ(v)：对于顶点 v，直接后继集合是指所有直接依赖于 v 执行结果的指令集合。即 {n | n ∈ V 且 (v, n) ∈ A}。

顶点的引入次数 in(v)：指顶点 v 的引入弧的数量，即 | {n | n ∈ V 且 (n, v) ∈ A} |。

1.1.2 基本块 DAG 的特性

1．至少一个顶点的引入次数为 0:这意味着至少有一条指令(根节点 root)不依赖于基本块中的任何其他指令，通常是基本块中的第一条指令。

2．相同的直接前驱但无依赖关系的顶点可并行执行:如果两个顶点 v1 和 v2 有相同的直接前驱集合且它们之间没有弧，那么它们可以按任意顺序执行，因为它们之间没有依赖关系。

3．对于基本块中的指令序列 v1,v2,...,vn，任何指令 vj (1 ≤ j ≤ n) 在图中不会有弧(vj, vi) (j > i)，这意味着指令 vj 只能在指令 vi 执行完之后执行。

1.2 算法步骤

(1) 初始化[ii]

创建一个空的指令序列 (schedule)，它将用于存储调度后的指令。

创建一个缓冲区 (Schedule buffer)，它将用于临时存储待调度的指令。

初始化一个时间数组(starttime[])，该数组用于记录每个指令可以开始执行的时间，初始值设为 0。

(2) 构建 DAG

遍历基本块的汇编指令，并构建一个 DAG 图来表示这些指令之间的依赖关系。

为 DAG 中的每个顶点（即指令）建立它的直接后继顶点集合。

如下图示，是选自卡内基梅隆大学（Carnegie Mellon University）的 Compiler Design 课程，它表示的是指令依赖关系图构建：

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

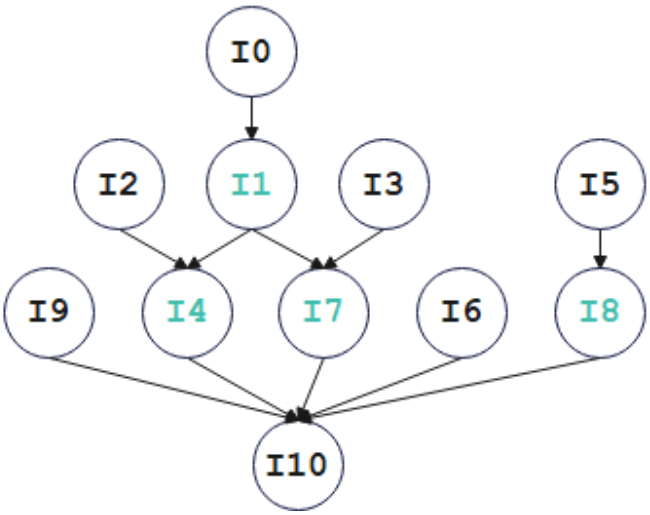


图 1-1 指令依赖图示意图

（3）找出根顶点

在 DAG 图中找出所有最先可以执行的指令，即那些没有前驱的顶点。

将这些根顶点放入缓冲区（buffer）中。

（4）调度循环

当缓冲区（buffer）不为空时，进行以下步骤：

4.1 排序：将缓冲区 buffer 中的顶点按其直接后继顶点可以开始的最小时间从大到小排序。

4.2 指令调度：从缓冲区（buffer）中取出一个顶点（即指令），将其从缓冲区中删除并添加到调度序列（schedule）中。

4.3 更新后继顶点：对于当前指令的每个直接后继顶点，将其可以开始执行的时间（starttime[j]）更新为成当前指令的执行时间（starttime[i]）加上从当前指令到后继指令的延迟（w(i，j)）和后继指令自身的延迟，取这个值和

starttime[j]的较大值。同时，将后继顶点的 in(j)（表示还需等待多少个前驱指令完成）减 1。

4.4 加入新顶点：如果某个后继顶点的 in(j)变为 0（即没有前驱指令需要等待），则将其加入缓冲区（buffer）中。

（5）当缓冲区为空时，说明调度完成，此时 schedule 数据结构中存储的就是调度后的指令序列。我们这时候需要遍历调度后的指令序列，对于任意两条相邻的指令 Ii 和 Ij，如果 starttime[j] > starttime[i] + 1， 则在 Ii 和 Ij 中加入 starttime[j]-starttime[i]-1 个 NOP 空操作来确保时序正确。

1.3 关于指令调度算法个人观点和结论

指令调度算法是编译器优化中不可或缺的一环。它通过重新安排指令的执行顺序，不仅减少了指令间的依赖关系，还提高了指令的并行执行能力，从而显著提升了程序的执行效率。这种优化对于现代高性能计算至关重要，尤其是在处理复杂计算任务和大数据时。

在这个指令调度算法中，我们使用 DAG 图作为指令间依赖关系的直观表示，为指令调度提供了强有力的支持。通过构建 DAG 图，编译器能够清晰地识别出指令间的约束条件，从而制定出更加合理的调度方案。此外，DAG 图的特性也为指令的并行执行提供了理论基础，使得编译器能够充分利用硬件资源，提高程序的执行效率。

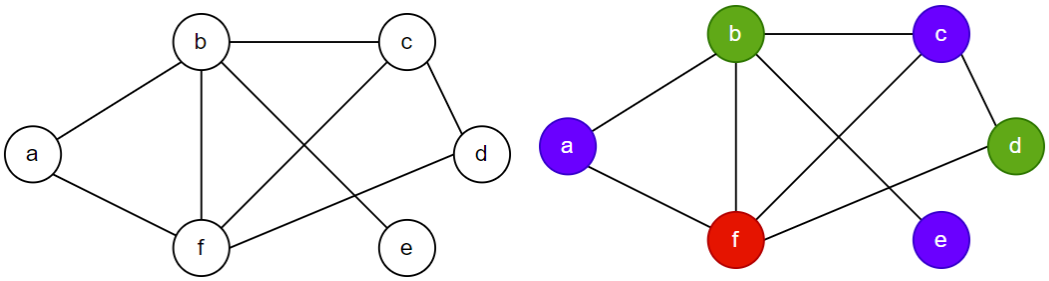
当然，这个算法是一个比较基础的指令调度算法，它属于静态调度的范畴。对于指令调度算法的探索和优化是一个持续不断的过程，我们还应该多多尝试其他一些更为精妙的指令调度算法，通过各种算法的搭配，取长补短，将指令调度优化到极致。比如，通过引入启发式规则，可以设计出更加高效和灵活的调度算

法，以应对复杂的程序结构和硬件环境；指令调度算法不仅要考虑程序的执行效率，还要考虑其他因素，如功耗、资源利用率等。因此，研究多目标优化的指令调度算法，通过权衡不同目标之间的关系，可以设计出更加全面和优秀的调度方案；随着机器学习技术的不断发展，我们也可以将其应用于指令调度领域。通过训练机器学习模型来预测指令的执行时间和依赖关系，可以为指令调度算法提供更加准确的指导。

2 图着色寄存器分配算法

2.1 图着色问题

图着色问题的核心思想为：有相邻关系的结点不可以被染成同一种颜色。如图 2-1 所示。



（a）　　（b）

图 2-1 三色图着色问题示意图[iii]

图 2-1(a)中一共 6 个结点，被边相连的结点表示有相邻的关系。假设一共有红绿蓝 3 种颜色可供选择，那么一个可能的染色的结果就为图 2-1(b)。

在这个三可着色图问题中，我们的染色过程可用是这样：每次将一个度低于 3 的节点以及它的边删除，并将该节点放入栈中。重复这个操作，直到所有结点都放入了栈中，此时图就为空图。然后，我们通过栈，将结点依次取出，并为其着色，如果原先图中的两个结点是邻接关系，则不能用同一个颜色。我们依次为

结点着色，直到栈空。这样一个着色问题就解决完毕。

2.2 利用图着色进行寄存器分配

分配寄存器也是类似的思路，图中的结点可以被看作是中间代码中的虚拟寄存器，而颜色可以被看作是物理寄存器，每个物理寄存器对应一种颜色，分配物理寄存器就是对结点进行着色。结点之间的边代表了两个虚拟寄存器的冲突，意味着这两个寄存器不能被分配到同一个物理寄存器中。

我们可用使用一些计算 live Interval 的方法[iv]来获得每个虚拟寄存器的 live interval 信息。而虚拟寄存器的边就是 live interval 产生了重叠，所以可用通过 live interval 画出对应的冲突图来。

如果两个虚拟寄存器的 live interval 刚好是前后重合，即如下图示：

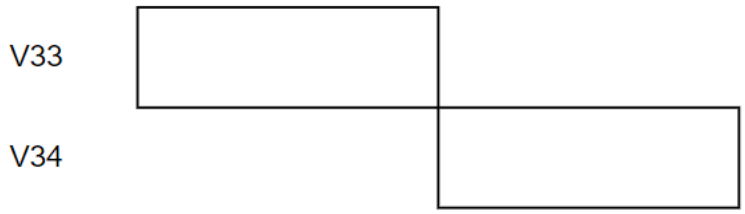


图 2-2 move-related 示意图

那么这种关系被称为移动关联 move-related。这两个虚拟寄存器实际上是可以合并成一个的，也就是染上同一种颜色。

但是，如果我们将两个区间合并成了一个区间之后，产生了更多的冲突，那么就得不偿失了。如下图所示，本来一个寄存器只有一个冲突，现在一个寄存器有两个冲突，这意味着冲突图中这一个结点的度变成了 2。

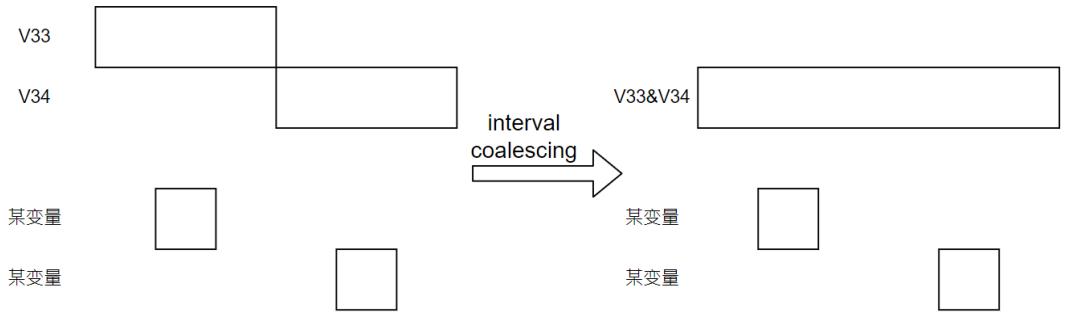


图 2-3 节点合并新增溢出示意图

并且，也不是每一对 move-related 结点都可以合并，考虑如下情形：A 和 B 为 move-related 关系，B 和 C 也是 move-related 关系，但 A 和 C 冲突，这样，不管是 A 和 B 合并还是 B 和 C 合并，都会产生冲突，因此不能合并。

寄存器分配时往往会出现一些不可着色图，这就需要进行溢出 spilling。如果一个结点的度大于可分配寄存器的数量，那么这个结点有可能无法分配到寄存器。

当结点无法分配到寄存器时，虚拟寄存器的值被保留在栈中，而不是物理寄存器里，只有在使用的时候才取出来。在被使用的时候需要指派一个寄存器来装载内存中的值，所以在使用点也需要被分配一个寄存器，还需要在 IR 中插入有关的存取指令，虽然依旧需要使用寄存器，但是实际上新加入的虚拟寄存器的活跃区间非常短，所以对寄存器分配产生的压力也小。

2.3 关于图着色寄存器分配法的个人观点和总结

基于图着色的寄存器分配法迭代过程如下所示[v]

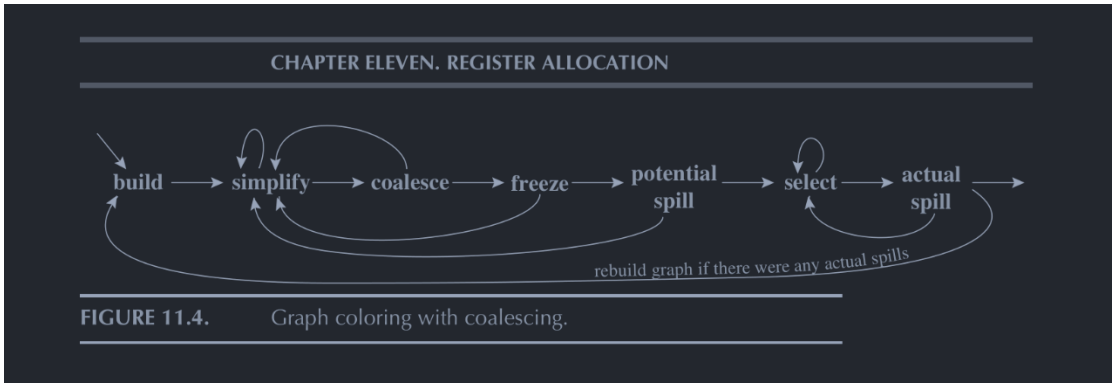


图 2-4 图着色寄存器分配迭代过程示意图

1． 在寄存器图着色分配的过程中，我们首先遍历指令，将虚拟寄存器用结点放的方式标注出来，并分析每个结点的冲突关系，构造出冲突图。

2． 然后，我们使用“简化”的方式，即每次将一个度低于可着色数的节点以及它的边删除，并将该节点放入栈中。重复这个操作，直到所有结点都放入了栈中，此时图就为空图。

3． 如果我们在简化过程中，发现简化进行不下去了，即有一结点度数大于可着色书，那么我们进行下一步。我们分析结点之间是否具有 `move-related` 关系，如果有，并且可用合并，则合并，否则，就将该关系撇除。在这个步骤中，我们尽量找那些度数较低的结点，因为这种结点出现溢出的可能性较小。

4． 接下来我们找出潜在的溢出结点，即度大于可着色数的结点，通过重新安排染色方案来减少真正的溢出结点。

5． 当我们找出真正的溢出结点时，要重新更新和该结点相关的结点，因为溢出结点只需要一小段寄存器时间，所以要将虚拟寄存器换名。

6． 我们从简化过程后的栈中选出结点进行染色，重复选择-染色这个过程，如果发现了有真正溢出的结点，那就将该结点记录下来，继续染色过程。在所有选择进行完之后，如果有真正溢出的结点，需要重新给代码分配虚拟寄存器，然

后重构冲突图，再循环上述步骤。一般两轮内就能结束染色。

基于图着色的寄存器分配方法不仅提供了一个理论上优雅的解决方案，而且在实际应用中证明了其高效性和实用性。这个方法能够有效地解决寄存器分配问题，特别是当物理寄存器数量有限时，通过冲突图和简化处理，能在很大程度上减少寄存器溢出情况。通过处理 `move-related` 关系和重新安排溢出结点，能够在不引入更多冲突的前提下优化寄存器使用。

通过不断优化和调整，该方法能够应对复杂的编译器优化需求，是编译器设计和实现中的重要工具。未来，随着计算机体系结构的发展和编译技术的进步，基于图着色的寄存器分配方法有望进一步完善和扩展，继续发挥其关键作用。

参考文献

[i] Cooper K D, Torczon L. Engineering a compiler[M]. Morgan Kaufmann, 2022.

[ii] 赵贤鹏, 李增智, 宋涛, 等. 一种基于 GCC 的 VLIW 编译器指令调度算法[J]. 微电子学与计算机, 2004, 21(1): 62-64.

[iii] Appel A W. Modern compiler implementation in C[M]. Cambridge university press, 2004.

[iv] Bloggs, j.(2022.02.22).计算 Live Interval[博客文章].取自 https://www.cnblogs.com/AANA/p/16311477.html

[v] George L, Appel A W. Iterated register coalescing[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1996, 18(3): 300-324.