12 寄存器分配-ARM

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆ 了解寄存器分配问题
- ⋆ 掌握 ARMv8-A 架构中主要通用寄存器的使用约定
- ⋆⋆⋆ 理解寄存器分配问题的图着色建模方法
- ⋆⋆⋆ 掌握基于单纯消除序列的寄存器分配算法

12.1 ARMv8-A 中的寄存器

ARMv8-A 的 AArch64 执行状态提供 31 个 64-bit 通用寄存器 `x0-x30`。当仅访问其低 32 bit 时，使用对应名称 `w0-w30`。尽管这些寄存器被称为通用寄存器，但在函数调用过程中仍需遵循 AArch64 调用规约（Calling Convention）。如表 12.1 所示，寄存器 x0-x1 用于参数和返回值传递，`x30` 用于保存函数返回地址，`x29` 一般用于保存栈帧基地址。另外，`x9-x15` 和 `x19-x28` 寄存器用途不限，但前者是 Caller-saved，后者是 Callee-saved。Caller-saved 表示寄存器的内容在函数调用过程中可能被被调用者修改。因此，如果调用者希望在调用结束后继续使用其中的值，则需要在调用前保存、调用后恢复。Callee-saved 表示被调用函数若使用该寄存器，则必须在使用前保存其原值，并在返回前恢复。

表 12.1: AArch64 中的寄存器用法约定

寄存器名称	调用规约	注释
x0-x1	参数 1-2/返回值	
x2-x7	参数 3-8	
x8	特殊用途：间接调用返回地址	
x9-x15	普通寄存器	Caller-saved
x16-x18	特殊用途	
x19-x28	普通寄存器	Callee-saved
x29	栈帧基地址	
x30	返回地址	

寄存器分配（Register Allocation）是编译器后端的重要翻译和优化阶段，其目标是将中间表示或目标代码中的虚拟寄存器映射到有限数量的物理寄存器上。该过程涉及寄存器预分配、通用寄存器分配和寄存器溢出环节。其中，某些虚拟寄存器由于调用规约或特定指令要求，只能映射到指定物理寄存器。这类约束通常在通用寄存器分配之前处理，因此称为预分配（Pre-allocation）。函数调用相关的寄存器分配可概括为以下几点：

- **参数和返回值传递**：LLVM IR 中的函数调用和返回均是由单条 IR 指令完成的，参数和返回值传递未考虑调用规约的问题，将其翻译为汇编代码时应：函数调用前先将参数按照顺序依次拷贝到 `x0-x7` 中；函数返回前将返回值拷贝到 `x0-x1` 中。

- **返回地址**：如果当前函数涉及一处或多处函数调用，应在函数入口处将 `x30` 寄存器内容保存到栈上，函数返回前将其还原。否则，Callee 会改写 `x30` 的值，导致无法正常返回。
- **其它调用规约**：如果使用 `x9-x15` 这类 Caller-saved 寄存器并涉及函数调用，应当在函数调用前将其备份，调用后还原；对于 `x19-x28` 这类 Callee-saved 寄存器，应当在使用前将其备份，函数返回前还原。

12.2 通用寄存器分配

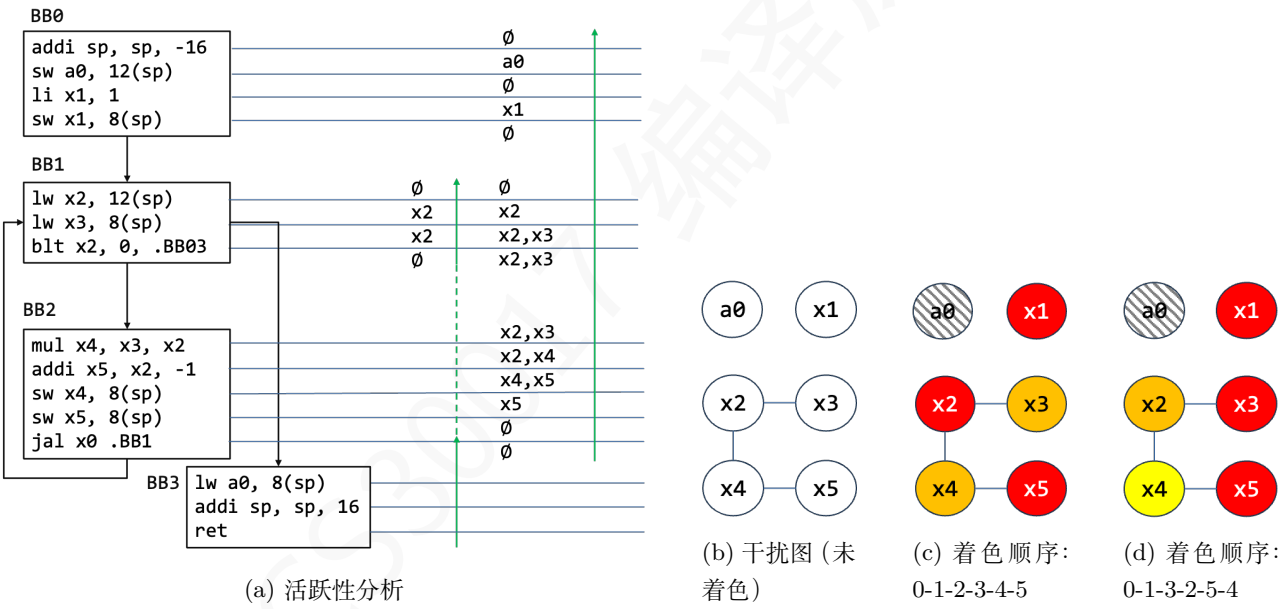
指令翻译时无需考虑虚拟寄存器数量限制，然而实际的物理寄存器数量是有限的。例如，AArch64 架构中一般使用 `x9-x15` 寄存器或 `x19-x28` 寄存器；X86 架构可用的寄存器则更少。因此，如何将虚拟寄存器翻译为物理寄存器非常关键。

下面我们使用干扰图对寄存器分配问题进行建模，并将其转化为着色问题。

12.2.1 干扰图构建

定义 1. 寄存器干扰图（Register Interference Graph）$\{V, E\}$ 是一个无向图，其中每一个点 $v_i \in V$ 表示一个虚拟寄存器，如果 $v_i, v_j \in V$ 同时活跃，则存在边 e_{ij}。存在干扰关系的虚拟寄存器应分配不同的物理寄存器。

干扰图是基于活跃性分析构建获得的，即分析每个虚拟寄存器的活跃区间。我们可以采用混沌迭代数据流分析方法：对控制流图进行逆序分析；在逆向数据流分析过程中，若一条指令使用了虚拟寄存器 v_i（即 use(v_i)），则认为 v_i 在该指令之前处于活跃状态；若该指令定义了 v_i（即 def(v_i)），则认为此前传播的活跃性在该点终止。遇到合并节点取并集即可。将同时存在活跃关系的虚拟寄存器两两相连便得到了干扰图。图 12.1a 和 12.1b 分别展示了一个活跃性分析示例及其干扰图构建结果。



(a) 活跃性分析　(b) 干扰图（未着色）　(c) 着色顺序：0-1-2-3-4-5　(d) 着色顺序：0-1-3-2-5-4

图 12.1: 干扰图构建和着色示例

12.2.2 着色问题

基于干扰图的寄存器分配问题是一个着色问题。

定义 2 (着色问题)**.** 对无向图 $\{V, E\}$ 的所有点 $v_1, ..., v_n \in V$ 进行着色，要求任意两个相邻顶点不能使用相同颜色。问题可以表述为：

1. 求该图的最小着色数（Chromatic Number）；
2. 给定整数 K，判断该图是否存在仅使用 K 种颜色的合法着色方案。

后者称为 K-colorability 问题，其中 K 个颜色代表 K 个物理寄存器。

K-colorable 问题在 $K \geq 3$ 时是 NP-Complete 的问题。

12.2.3 着色算法

我们假设存在颜色数组 Color = [红, 澄, 黄, 绿, 青, 蓝, 紫]，每次着色均采用当前编号最低的可能的颜色，即算法 1 。则着色问题本质上是着色顺序选取的问题。如图 12.1c 和12.1d 分别对应两种着色顺序，一种需要使用两个颜色，另外一种则需要使用三个颜色。

算法 1 颜色选取方法

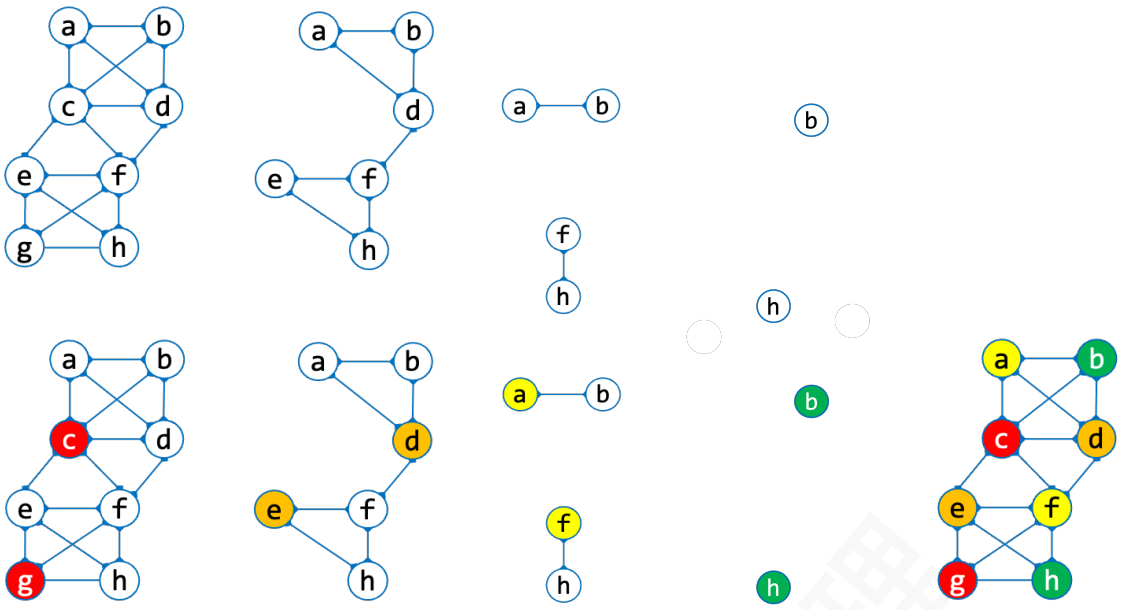
procedure GreedyColoring($G(V,E), \pi$) ▷ $\pi = (v_1, v_2, \ldots, v_n)$ is a vertex ordering
  let $C = \{c_0, \ldots, c_{K-1}\}$
  for each $v_i \in \pi$ **do**
    choose the smallest color c_j not used by any colored neighbor of v_i
    if such c_j does not exist **then**
      return FAIL
    end if
    $\mathrm{Col}(v_i) \leftarrow c_j$
  end for
  return Col
end procedure

针对着色问题有大量的贪心算法研究。比如线性扫描算法采用先到先得的思想，对于先遇到的寄存器先分配颜色。该方法的优点是实现简单且速度较快，无需维护干扰图的边数信息。另外还有一些基于干扰图边数选取着色顺序的启发式算法，下面介绍一种经典的递归最大优先（RLF：Recursive Largest First）算法 [1]。

算法 2 RLF 算法

procedure RLF($G(V,E)$)
  if $V = \emptyset$ **then**
    return
  end if
  let S be an empty set
  Find $v_i \in G$ with the maximum degree
  $S \leftarrow S \cup \{v_i\}$
  Let T be the set of vertices in G that are non-adjacent to every vertex in S
  while $T \neq \emptyset$ **do**
    Find $v_j \in T$ with the maximum degree
    $S \leftarrow S \cup \{v_j\}$
    Update T
  end while
  Assign a new color to all vertices in S
  Remove S from G
  RLF(G)
end procedure

RLF 的具体实现如算法 2 所示。该方法分为多轮次递归进行，每一轮首先选取当前图中度数最大的顶点作为种子顶点，并赋予一个新的颜色。随后，在所有与当前已选顶点均不相邻的顶点中，继续选择度数最大的顶点赋予相同颜色，直到无法再加入新的顶点为止。



(a) 第 1 轮　(b) 第 2 轮　(c) 第 3 轮　(d) 第 4 轮　(e) 结果

图 12.2: 应用 RLF 算法进行着色示例

图 12.2展示了应用 RLF 算法进行着色的过程。第 1 轮首先选取度数最大的顶点 c（度数为 5）作为种子顶点。在与 c 不相邻的顶点 g（度数为 3）和 h（度数为 3）中，任选 g 加入当前颜色类。此时已不存在同时与 c 和 g 均不相邻的顶点，因此第 1 轮结束，将 c 和 g 着为同一颜色，并从图中移除。第 2 轮在剩余图中选取度数最大的顶点 d（度数为 3）作为种子顶点。在与 d 不相邻的顶点 e 和 h 中任选 e 加入当前颜色类。由于不存在同时与 d 和 e 均不相邻的顶点，因此第 2 轮结束，将 d 和 e 着为同一颜色，并从图中移除。随后在剩余图上重复上述过程，直至所有顶点均完成着色。

12.2.4　基于单纯消除序列着色

下面介绍一类特殊的着色问题：即当虚拟寄存器满足 SSA 形式时，则该着色问题不是 NP-hard 问题。我们可以采用基于单纯消除序列的方法选取最优着色顺序，确定最优着色方案 [2]。

下面首先定义单纯消除序列的相关概念。

定义 3 (单纯点（Simplicial Vertex）)**.** 无向图 $\{V, E\}$ 中的如果顶点 v_i 的所有邻居两两相邻，即其邻居集合诱导出的子图为一个团（Clique），则称 v_i 为单纯点。

定义 4 (完美消除序列（Perfect Elimination Ordering）)**.** 若按照某一顶点序列依次删除顶点，且每次被删除的顶点在当前剩余图中均为单纯点，则称该序列为完美消除序列。

定义 5 (单纯消除序列（Simplicial Elimination Ordering）)**.** 完美消除序列的逆序称为单纯消除序列。

定义 6 (弦图（Chordal Graph）)**.** 若无向图中任意长度大于 3 的简单环都至少包含一条弦（即连接环上两个非相邻顶点的边），则称该图为弦图。

SSA 形式程序构造出的寄存器干扰图一定是弦图。因此，可以利用弦图的结构性质在线性时间内获得最优着色方案，从而避免求解一般图着色问题的 NP-hard 性。弦图一定存在单纯消除序列。给定一个弦图，找单纯消除序列可以采用最大势算法，具体可参考算法 3 。

算法 3 最大势算法

```
1: procedure MCS(G(V, E))
2:     for each v_i ∈ V do
3:         w(v_i) = 0;
4:     end for
5:     W = V;
6:     for each i ∈ [1..n] do
7:         Let v be a node with max weight in W
8:         for each u ∈ Neighbor(v) do
9:             w(u) = w(u) + 1
10:        end for
11:        W = W - v;
12:    end for
13: end procedure
```

表 12.2: 应用最大势算法找图 12.2e 的单纯消除序列。

步骤		**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
0	初始化	0	0	0	0	0	0	0	0
1	选取 a		1	1	1	0	0	0	0
2	选取 b			2	2	0	0	0	0
3	选取 c				3	1	1	0	0
4	选取 d					1	2	0	0
5	选取 f					2		1	1
6	选取 e							2	2
7	选取 g								3
8	选取 h								

表 12.2 展示了应用最大势算法求解图 12.2e 的单纯消除序列的过程。算法初始时所有顶点权值均为 0，每一步选择当前权值最大的未访问顶点加入序列，并将其所有未访问邻居的权值加 1。对于该图，MCS 依次选取顶点 `a,b,c,d,f,e,g,h`，从而得到完美消除序列 `(a,b,c,d,f,e,g,h)`，其逆序 `(h,g,e,f,d,c,b,a)` 即为对应的单纯消除序列。由于该图是弦图，按照该单纯消除序列进行贪心着色能够得到最优着色结果。

12.3 寄存器溢出

当干扰图所需颜色数超过可用物理寄存器数量时，部分虚拟寄存器无法映射到物理寄存器，此时需要将其值临时存放到内存中，这一过程称为寄存器溢出（Register Spilling）。发生溢出后，每次使用该值时都需要插入额外的加载指令，每次更新该值时则需要插入存储指令。因此，寄存器溢出通常会显著增加程序运行开销。实际编译器通常根据寄存器的使用频率、活跃区间长度、所在循环深度以及图结构特征等信息估计溢出代价，并优先选择代价较低的寄存器进行溢出。

参考文献

[1] Frank Thomson Leighton, “A graph coloring algorithm for large scheduling problems.” Journal of Research of the National Bureau of Standards, 1979.

[2] Fernando Magno Quintao Pereira, and Jens Palsberg. “Register allocation via coloring of chordal graphs.” Asian Symposium on Programming Languages and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

练习

1) 构造一个非弦图，使得 RLF 算法无法找到最优解。

2) 如果一个图是弦图，RLF 算法是否一定能找到最优解?