Introduction to optimal PRE algorithms

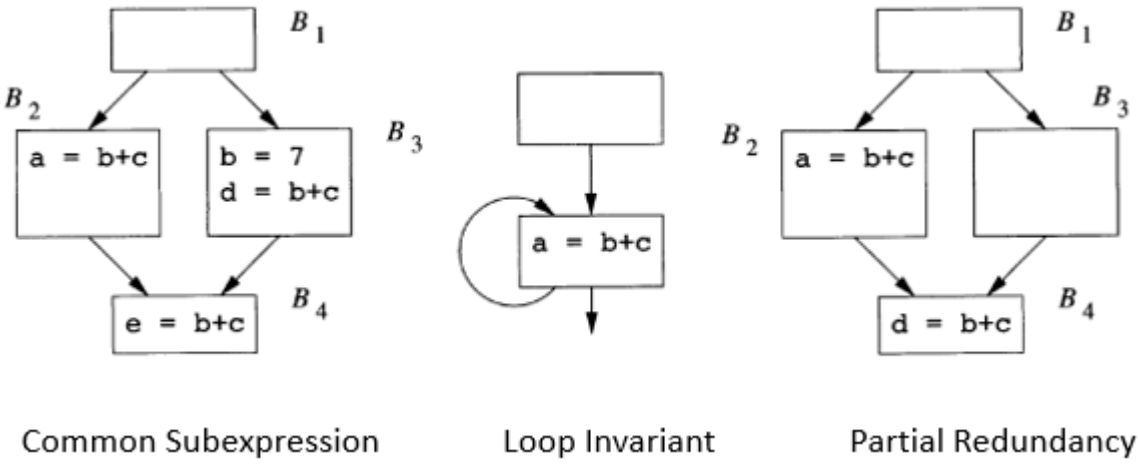
宋小牛 PB15000301

Partial Redundancy Elimination

冗余计算是程序中很常见，又很难被避免的一种存在。冗余一般有这些来源：

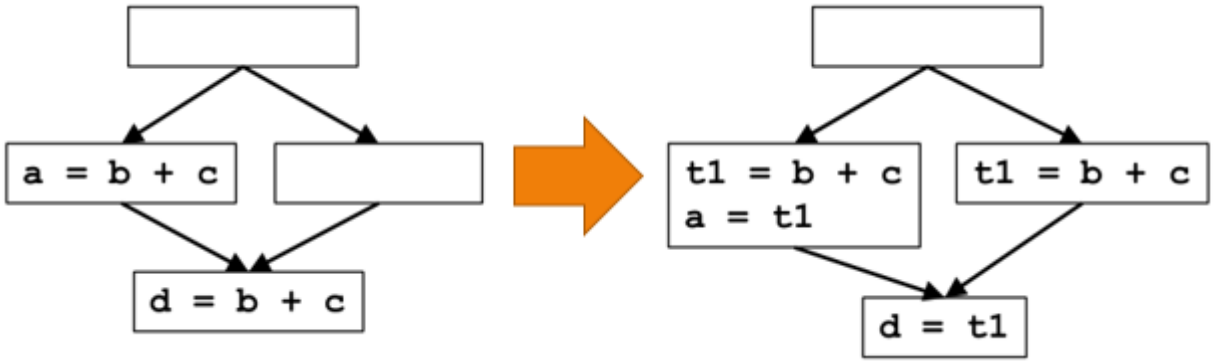
- *高级语言翻译至低级语言产生的冗余计算*：在高级语言中可能并不会观察到荣誉计算，但在生成的中间表达式，乃至汇编上也可能出现冗余计算
- *为了便于维护、阅读而刻意设置的冗余计算*：处于软件工程上的考虑，代码需要方便的维护，需要使用较为清晰的逻辑来展示代码，而消去荣誉计算会使代码难以理解
- *模板、程序框架导致的冗余*
- *程序员有限的视角*：程序员在编程时关注于局部代码的细节，不一定能考虑到全局上的冗余计算

对于程序员来说，冗余计算一般有如下这三种形式：*公共子表达式(Common Subexpression)*、*循环不变量(Loop Invariant)*、*部分冗余(Partial Redundancy)*，他们的例子如下：



实际上，在定义中我们将冗余分为两类：**完全冗余与部分冗余。**表达式E在程序点p处完全冗余是指，从程序入口处到p的**所有路径**上都会出现E的冗余计算，而部分冗余是指仅在一部分路径上会出现E的冗余计算。上图中的公共子表达式就是完全冗余，而循环不变量实际上是部分冗余的一种特例：b+c可能被执行1一次，也可能被执行很多次.

完全冗余可以通过公共子表达式删除（CSE）来完全消去，而部分冗余只能通过专门的Partial Redundancy Elimination（PRE）算法消除，其基本操作一般有：在新的位置插入计算；创建新的Basic Block；复制Basic Block。下图是一个PRE的例子：

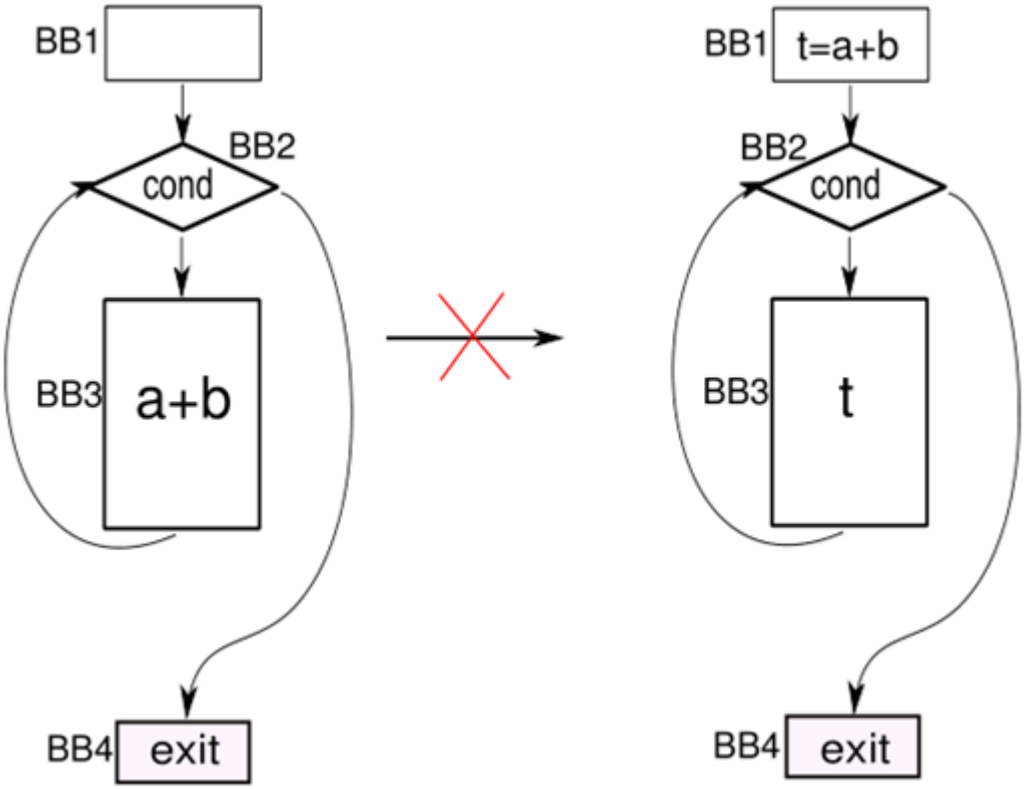


Safe PRE：

在进行PRE时，插入的表达式计算的位置一定要是**安全**的：也就是说要保证插入的计算相比原来程序不会增加原来不出现的计算。这样能保证程序至少不比原来慢，同时不会因为新插入的计算而导致异常退出。保证安全性的插入方式是：在q位置插入E的计算，必须要从q到程序出口处上的每条路径都会出现E的计算。

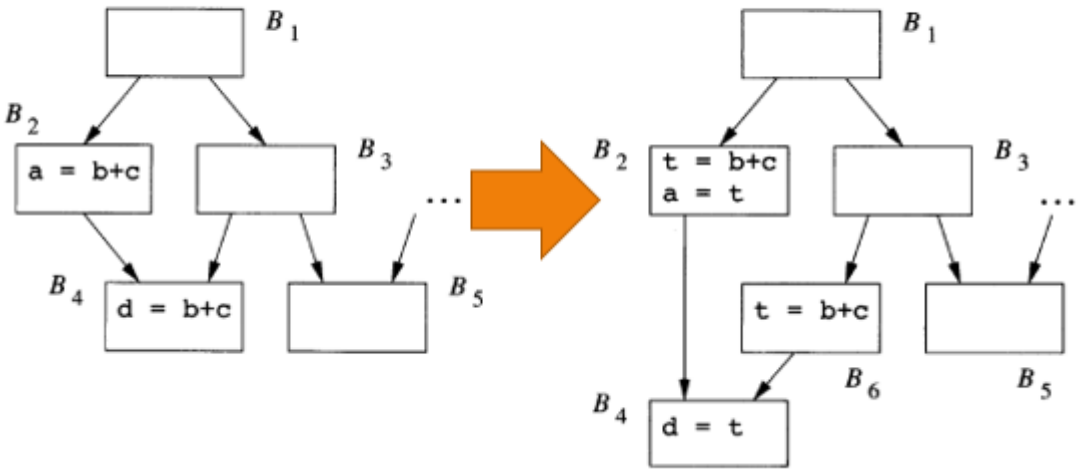
安全性是一个很强的性质，不是所有的部分冗余都能被它消除。

While-Do循环中的循环不变量是一个典型的不能安全消除的例子：如下图，如果在BB1中插入a+b的计算，可能出现的程序路径BB1-BB2-BB4上新增加了一个原本不会出现的计算。

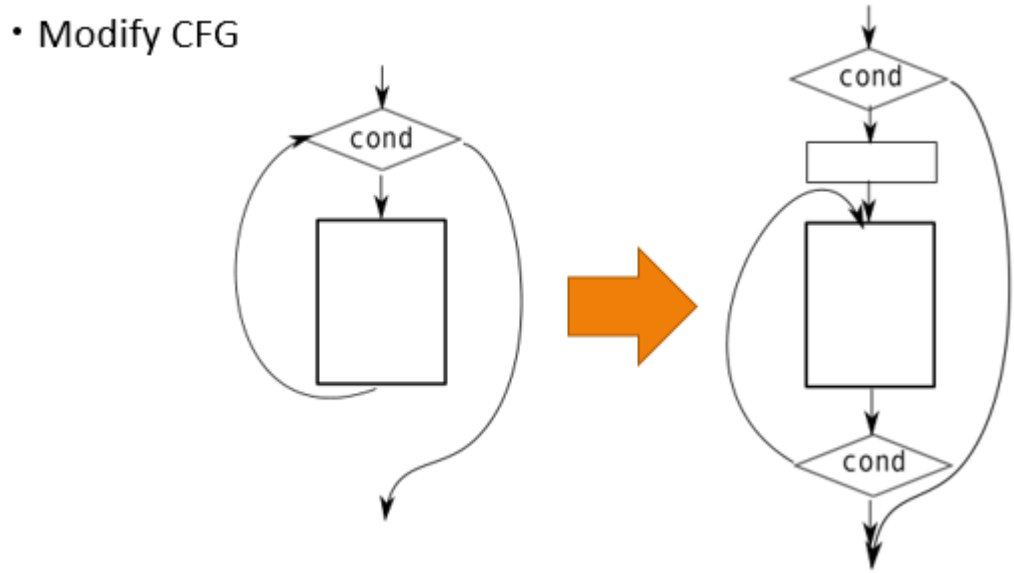


前面提到除了直接插入表达式，有时还需要变换程序控制流图来进行安全的PRE。最基本的，在关键边上插入一个新的Basic Block，如下图：

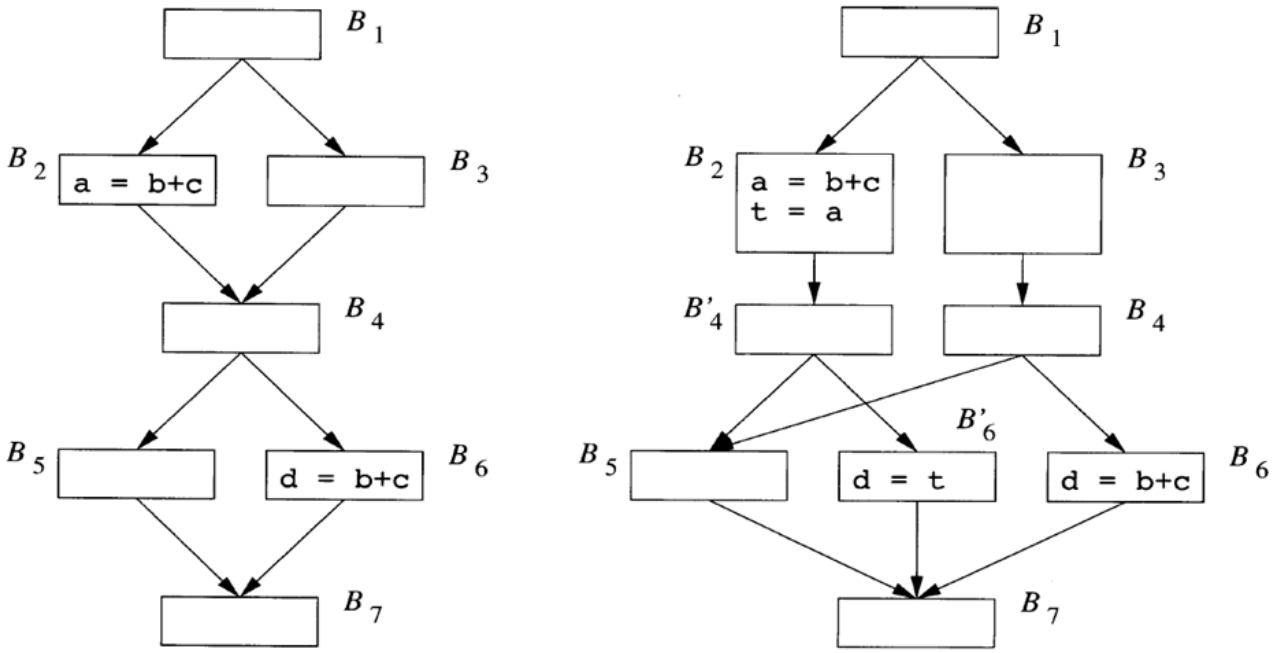
- Modify CFG



前面提到的While-Do循环也可以通过变换流图的方法消去循环不变量：将While-Do变为Do-While，之后即可提出循环不变量：



但有些情况不能通过简单的插入新块来解决，只能复制基本块：



但是基本块数量与冗余路径上分支结点的个数是指数关系，这将非常限制的提高程序大小，并且显著降低cache命中率，因此一般不考虑复制代码块的方法。

Lazy Code Motion

LCM是一个经典的最优PRE算法，他能达到*计算最优*以及*生命周期最优*。

- 计算最优：优化后的结果不能被其他任何安全算法降低表达式的计算次数
- 生命周期最优：计算最优的同时，优化后引入的临时变量的声明周期最短

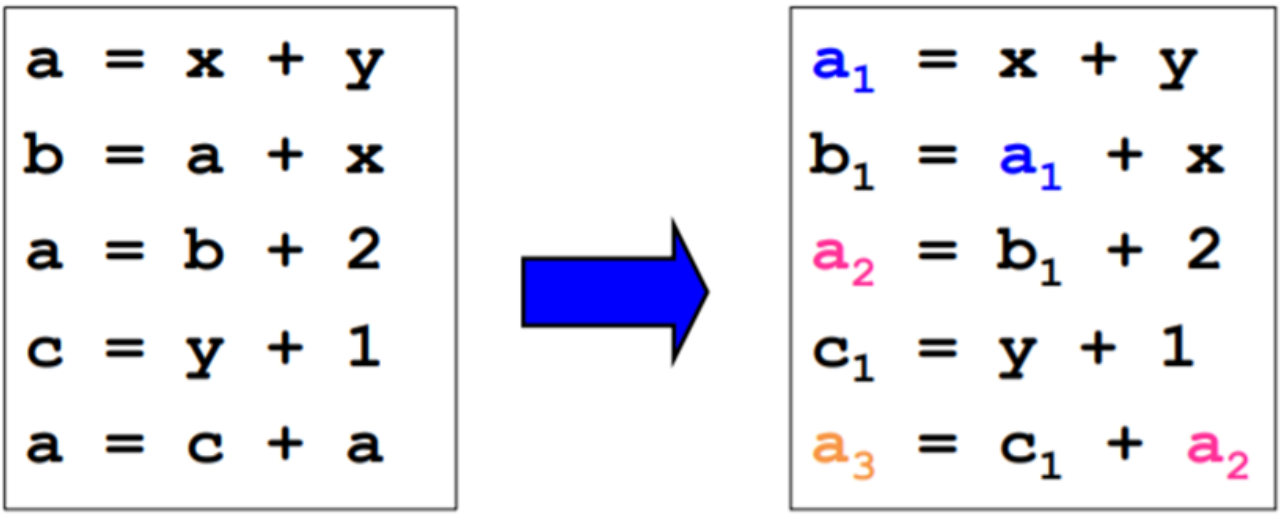
LCM通过4次数据流分析（如下）来解决PRE问题：首先找到最早的可以安全插入表达式计算的Basic Block，这样可以消除尽量多的冗余计算；再将这些表达式的计算尽可能延迟，从而使得引入的临时变量生命周期最短。

1. Anticipated Expressions 预期表达式
2. Available Expressions 可用表达式
3. Postponable Expressions 可延迟表达式
4. Used Expressions 引用表达式

类似LCM这样的基于传统数据流分析的算法称为是基于bit-vector的，因为在数据流分析时其对每个Basic Block都维护一个bit-vector，每个bit对应一个程序中的表达式

Static Single Assignment

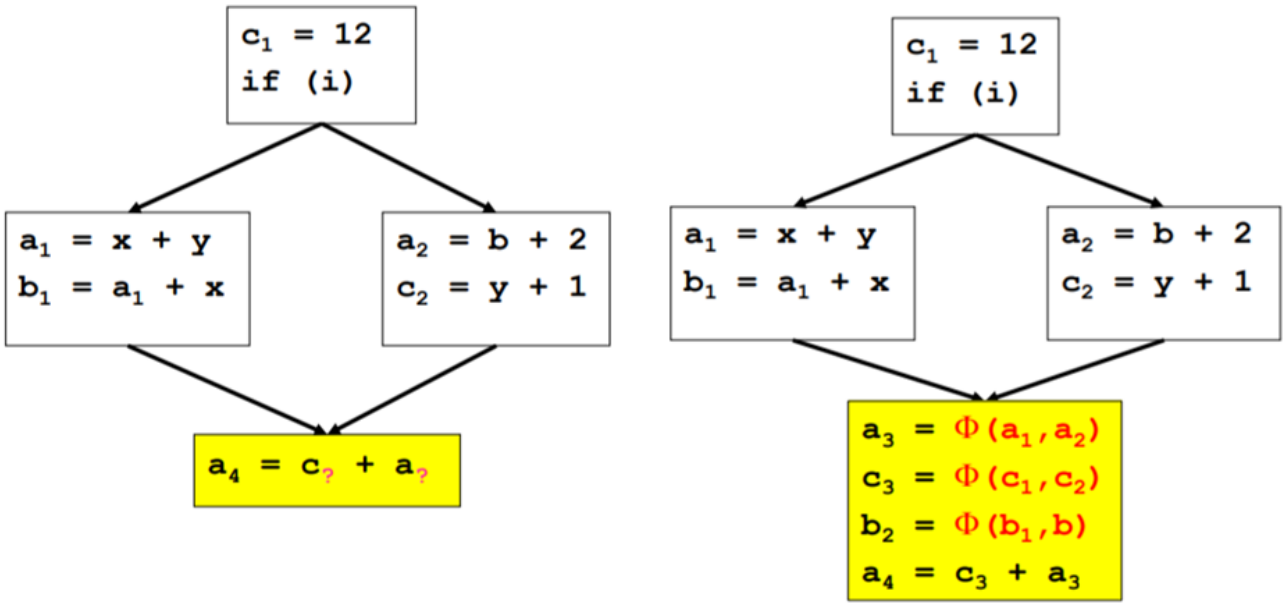
SSA(Wiki)是一种程序的中间表达，在这个中间表达中，每个变量都只被赋值最多一次。常用的将控制流图转化为SSA形式的方法，是在赋值语句将旧的变量名加上一个下标（如图）：



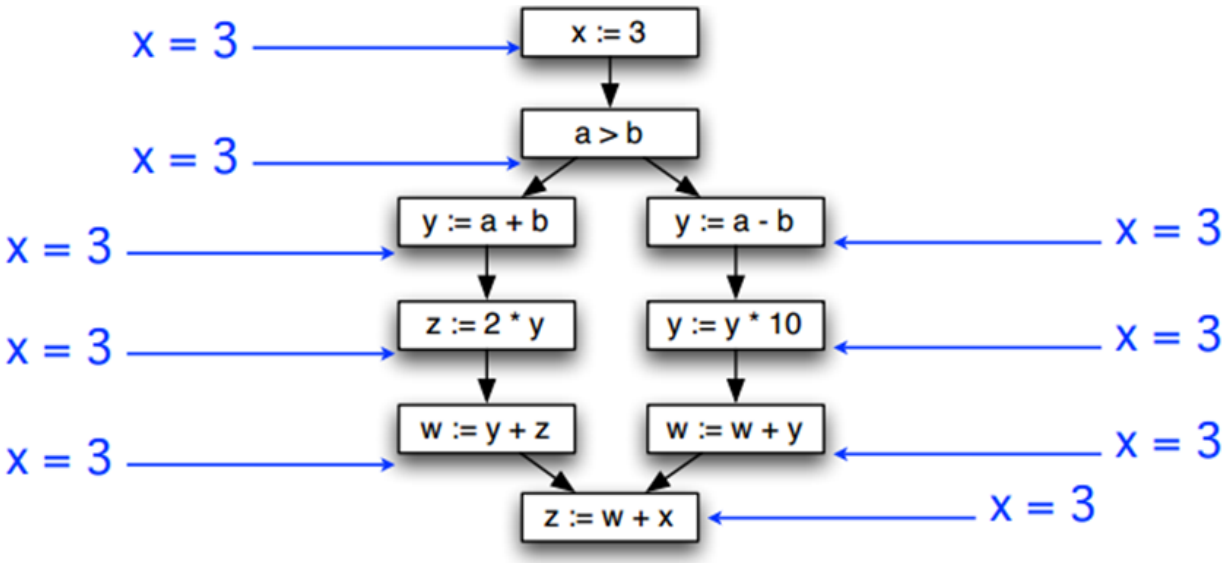
通过SSA可以实现很多优化

- 常数传播 （constant propagation）
- 值域传播（value range propagation）
- 稀疏有条件的常数传播 （sparse conditional constant propagation）
- 消除无用的程式码 （dead code elimination）
- 全域数值编号 （global value numbering）
- 消除部分的冗余 （partial redundancy elimination）
- 强度折减 （strength reduction）
- 暂存器配置 （register allocation）

转换为SSA时会出现一个块的多个前驱块里有y的不同定义的情况，分别为$y_1, y_2, \ldots, y_n$，这时候要在块中使用变量 y 的话需要决定使用哪一个传入的 y。在这个块的开头中插入一个Φ表达式$y_{n+1} = \Phi(y_1, y_2, \ldots, y_n)$，当程序从第$j$个前驱进入这个基本块，$\Phi(y_1, y_2, \ldots, y_n) = y_j$。



为什么我们需要SSA呢？在传统的数据流分析中，bit-vector意味着我们要携带所有的分析信息经过每一个程序点。如果程序中有很多的赋值，代码量很大，这样的分析将带来很大的开销。而且在大部分情况下，我们在每个程序位置记录的是与该处无关的信息：



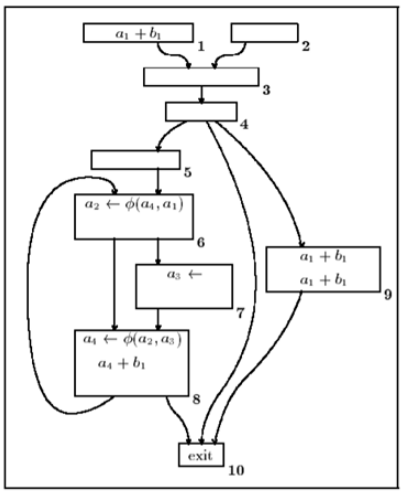
而SSA相当于定义了一个较为稀疏的def-use关系，显式的将值传递的关系体现出来，更快的实现更多优化。

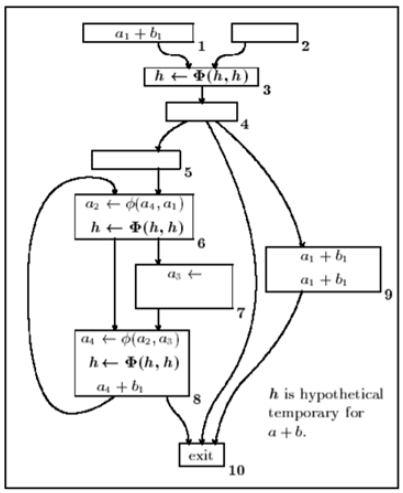
现代编译器大多使用SSA作为他们的中间表达。

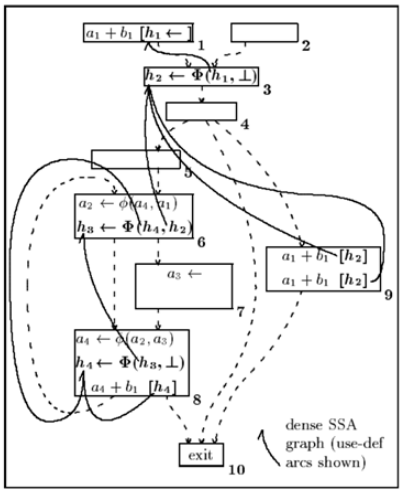
SSAPRE

SSAPRE是一个基于程序的SSA形式进行PRE的算法，在此之前的PRE算法使用的都是基于bit-vector的数据流分析。注意到，之前提到的SSA都是针对变量有def-use关系，Fred Chow则将表达式的冗余关系与其使用-定义关系联系起来，从而建立表达式的SSA形式，称为FRG（Factored Redundancy Graph），并提出SSAPRE算法。下图是一个建立FRG的例子。

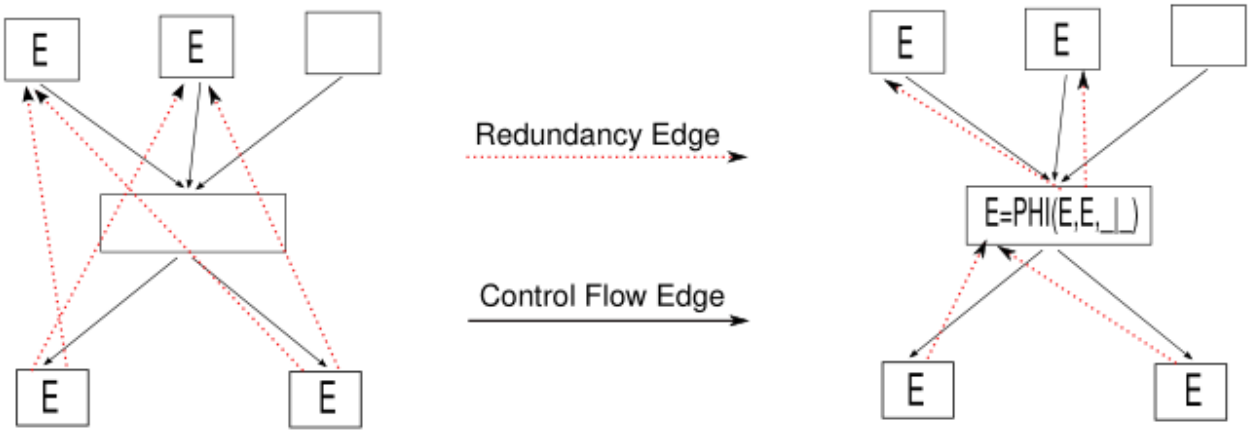
注意这里相比SSA不同的是，变量的phi操作数一定有对应的支配定义，而表达式的Phi操作数可能不是冗余的：Φ表达式的多个前驱中可能有某个没有这一表达式的计算，这时候我们将其记为⊥







简化的示意图如下



SSAPRE的基本步骤如下：

1. Φ - Insertion
2. Rename
3. Down safety
4. Will Be Avail
5. Finalize
6. Code Motion

前两步建立表达式的SSA形式；中间两步决定在安全PRE的情况下，如何插入新的计算；第五步引入新的计算记为一个临时变量，并对图中计算判断其应该引用之前的临时变量还是在本地计算再存为临时变量；第六步将做的修改更新为SSA形式

Speculative PRE

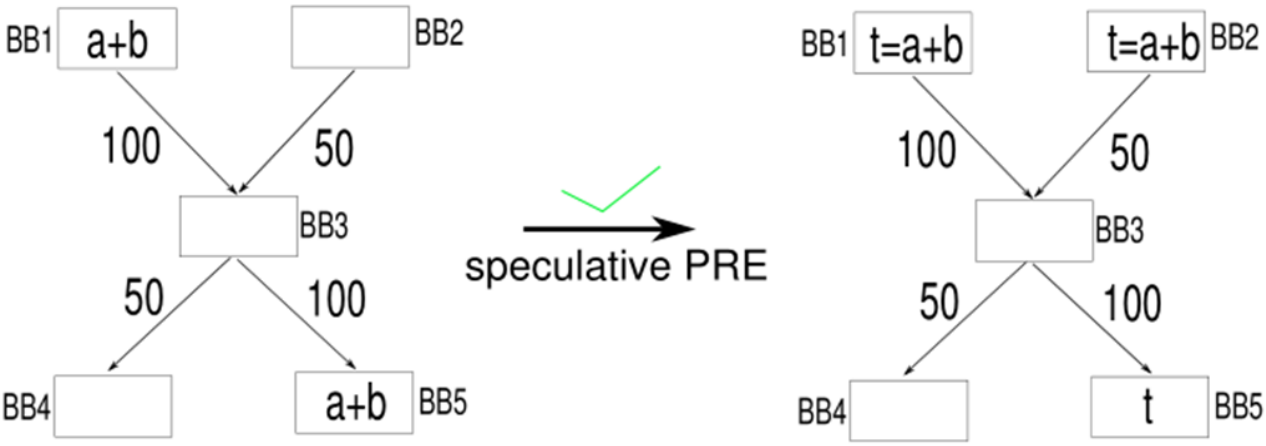
前面提到的安全PRE的安全限制条件较强，在不允许赋值代码块的情况下，有很多部分冗余的情况不能被消除；即使允许，也会显著增加代码大小。**Speculative PRE**是指：放松部分安全限制，在一些“不安全”的位置也允许插入表达式计算。因为不是所有表达式都会产生运行时异常，并且在程序实际执行时，流图上的边执行频次会有不同，Hot路径上可能有很多冗余计算，如果能获取程序在每条路径上的执行次数，那么在程序中插入原本不会出现的计算也不会导致运行速度相比原程序减慢。

通过**合规性测试（Legality Testing）**，我们可以确定插入的表达式是会在运行时产生新的异常。

现代编译器提供了基于剖析的反馈编译模式：

1. 第一遍编译，通过编译器插桩，或基于硬件计数器采样获得剖析信息
2. 第二遍编译，读入保存的剖析信息，得到块或者边的执行次数

通过这些信息，我们可以判断插入的新计算会不会导致计算次数增加。这样，我们可以消除程序中所有的冗余计算。



MC-PRE

MC-PRE是一个计算最优、生命周期最优的前瞻PRE算法。这里的计算最优与Safe PRE不同，是指基于剖析信息能达到的计算最优。

MC-PRE利用图的最小切割来找到最优插入点。但它是基于传统数据流分析的。现代编译器大多基于SSA形式，使用bit-vector形式的MC-PRE算法需要先消除其中的ϕ表达式，再将结果转换回SSA形式，这是十分低效的。

MC-SSAPRE

MC-SSAPRE是一个基于SSA的Speculate PRE最优算法。他的思路类似于SSAPRE，建立表达式的FRG，再利用MC-PRE的最小切割的思路进行PRE，最后用SSAPRE的方法完成算法

参考文献

PRE

- Morel, Etienne, and Claude Renvoise. Global optimization by suppression of partial redundancies. Communications of the ACM 22.2 (1979): 96-103.

LCM

- J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 224–234, 1992
- J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.

Safe PRE

- K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics*, 3(2 and 3):117–130, 1972.

MC-PRE

- Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the 2th annual IEEE/ACM international symposium on Code generation and optimization*, pages 91–102, 2003.

SSAPRE

- R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.
- F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, 1997.

SSA

- R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925

MC-SSAPRE

- Hucheng Zhou, Wenguang Chen, Fred C. Chow. An SSA-based algorithm for optimal speculative code motion under an execution profile. *PLDI* 2011