Распределение регистров, задача назначения переменных и временных значений физическим регистрам процессора, широко известна как одна из наиболее важных оптимизаций для компиляторов. Основной целью является минимизация трафика между оперативной памятью и процессором. Пропускная способность памяти часто является узким местом современных компьютерных систем, поскольку современный процессор работает намного быстрее, чем подключенная к нему основная память. Даже при наличии иерархии кэшей, обеспечивающих более быстрый доступ к часто используемым областям основной памяти, обращение к регистру в несколько раз быстрее, чем загрузка значения из памяти.

В большинстве процессорных архитектур регистры являются ограниченным ресурсом. Например, архитектура Intel IA32, описанная в рассматриваемой диссертации, предлагает только восемь регистров общего назначения, из которых только шесть могут быть использованы в обычных вычислениях. Поэтому в регистрах можно хранить только наиболее часто используемые значения. Правильное использование регистров имеет решающее значение для общей производительности программы.

Все значения, которые не могут быть сохранены в регистре, должны быть сохранены в стеке до того, как регистр будет перезаписан другим значением. Этот процесс называется переливанием. Когда выгруженное значение используется позже, оно должно быть снова загружено в регистр из памяти.

О важности распределения регистров можно судить по большому количеству алгоритмов, которые доступны сегодня. Грубая классификация выделяет два вида алгоритмов:
- Локальные методы ограничивают представление об алгоритме небольшой частью скомпилированного в данный момент метода. Внутренние циклы идентифицируются и оптимизируются.
- Глобальные методы пытаются оптимизировать целые методы или даже группы методов. Они могут достичь наилучшего результата, но работают значительно медленнее.

Имея достаточные знания о целевой архитектуре, можно было бы вычислить оптимальное распределение регистров, при котором время выполнения будет как можно меньше. Однако доказано, что оптимальное глобальное и даже локальное распределение регистров является NP-полной проблемой, которая не может быть использована на практике. В результате каждый алгоритм должен найти компромисс между временем компиляции при распределении и временем выполнения результирующего кода. Чем больше времени

затрачивается на выделение, тем быстрее генерируемый код. Этот компромисс особенно важен для компиляторов just-in-time (JIT), где время, необходимое для компиляции, является частью общего времени работы приложения. Поэтому многие JIT-компиляторы используют локальные методы, хотя существуют и глобальные методы, которые генерируют лучший код, но требуют слишком много времени на компиляцию.

Алгоритм линейного сканирования

Алгоритм линейного сканирования был впервые описан М. Полетто и др. в [Poletto97], когда они реализовали систему для динамической генерации кода. Алгоритм является очень быстрым, поскольку распределение выполняется за один линейный проход по интервалам времени жизни. Мы рассмотрим основную идею этого алгоритма. Также есть улучшенная версия, называемая second chance binpacking, была описана О. Траубом и др. Этот алгоритм тратит больше времени для получения лучшего распределения, например, он учитывает дыры в интервалах времени жизни и позволяет разбивать интервалы времени жизни на части во время распределения.

Базовый алгоритм линейного сканирования

Алгоритм линейного сканирования сначала упорядочивает все инструкции метода в линейном порядке, где все структуры потока управления, такие как условия и циклы, скрыты. Затем вычисляются интервалы времени жизни для всех виртуальных регистров. Каждый интервал начинается с момента первого определения регистра и заканчивается его последним использованием. Для учета влияния циклов и условий необходим анализ потока данных. Расчет интервалов времени жизни очень консервативен: поскольку дыры не допускаются, регистры считаются непрерывно живущими от первого определения до их последнего использования.

Алгоритм линейного сканирования работает непосредственно со списком интервалов, отсортированных по их начальным позициям. Компилятор выполняет итерации по списку и сразу же присваивает интервалу физический регистр. Если ни один физический регистр не доступен для всего времени жизни, то некоторые интервалы должны быть выведены в память. Два интервала времени жизни пересекаются, если их диапазоны пересекаются. Поэтому двум интервалам, которые не пересекаются, может быть назначен один и тот же физический регистр.

На каждом шаге алгоритм ведет список, называемый активным списком, который содержит все интервалы, перекрывающиеся с текущей позицией и

имеющие уже назначенные регистры. Интервалы, которые закончились еще до текущей позиции, удаляются из активного списка, поскольку они больше не актуальны. Интервалу, начинающемуся в текущей позиции, присваивается физический регистр, который не используется ни одним интервалом в активном списке. Если все регистры уже используются, один интервал должен быть пролит - и это либо интервал из активного списка, либо текущий обрабатываемый интервал. Оказалось, что хорошей эвристикой является выливание интервала с самой высокой конечной позицией.

Рассмотрим пример, который должен быть обработан с двумя физическими регистрами r1 и r2. Из-за консервативного подхода интервал времени жизни регистра v1 является непрерывным от инструкции (1) до инструкции (7).

В этом примере интервалы обрабатываются в порядке v1, v2, v3, v4, v5. Алгоритм начинается с пустого активного списка. На первом шаге обрабатывается интервал v1. Поскольку активный список пуст, первый физический регистр r1 присваивается v1, и v1 добавляется в активный список. Когда на следующем шаге обрабатывается v2, v1 все еще активен, поэтому r2 назначается на v2. Затем v2 добавляется в активный список.

Далее обрабатывается интервал v3. Поскольку активный список уже содержит v1 и v2 с назначенными физическими регистрами r1 и r2, для v3 нет физического регистра, и один интервал должен быть пролит. Алгоритм выбирает v1 для разлива, поскольку он имеет самую высокую конечную позицию, и удаляет его из активного списка. Ячейка памяти, которая назначается на v1, называется mem1. Регистр r1 больше не заблокирован и может быть присвоен v3, который добавляется в активный список.

Когда обрабатывается v4, конец v2 уже достигнут, поэтому v2 удаляется из активного списка. Теперь r2 не используется и может быть назначен на v4. Когда выделяется v5, никакие другие интервалы не активны, поэтому r1 может быть назначен. Теперь всем интервалам назначен регистр или ячейка памяти, и алгоритм останавливается.

Здесь показан код, в котором виртуальные регистры заменяются выделенными им физическими регистрами в соответствии с примером.

Полученное распределение не так уж и хорошо, потому что один виртуальный регистр должен быть выведен в память. Это является следствием консервативного построения интервалов времени жизни без дыр. Эта особенность компенсируется гораздо более быстрым временем

распределения. Поскольку требуется только один линейный проход по интервалам времени жизни, алгоритм линейного сканирования имеет асимптотическую временную сложность O(n), где n - количество виртуальных регистров.

Улучшенный алгоритм линейного сканирования (second chance binpacking)

Бинпакинг второго шанса - это расширение базового алгоритма линейного сканирования, которое производит более качественный код и в основном сохраняет линейную временную сложность. Одним из основных недостатков базового алгоритма линейного сканирования является тот факт, что он не допускает дыр в живых диапазонах. Особенно сложные графы потока управления имеют тенденцию создавать дыры из-за условий и циклов. Даже в простом примере, представленном в последней главе, интервал v1 имеет дыру от инструкции (3) до (5). Поскольку эта дыра игнорируется основным линейным алгоритм сканирования, интервал v1 должен быть разлит по памяти. Бинпакинг по второму шансу способен обрабатывать дыры в интервалах времени жизни.

Другим расширением бинпакинга второго шанса является возможность разделения интервалов: когда нет доступных регистров, базовый алгоритм линейного сканирования выводит весь интервал в память. Таким образом, весь интервал выгружается, даже если регистр доступен для части интервала. А бинпакинг второго шанса решает эту проблему путем разбиения интервалов: интервал начинается в регистре, но затем разделяется и разливается, если регистр больше не доступен. Также возможно, что высыпанный интервал будет перезагружен в другой регистр позже, он получает второй шанс разместиться в регистре.

Разделение интервалов приводит к гораздо лучшему использованию регистров, но также имеет некоторые недостатки. Поскольку линейное упорядочивание блоков не учитывает реальный поток управления, необходим второй проход, называемый разрешением. Инструкции перемещения вставляются на границах потока управления, когда интервалу назначено несколько мест. Если, например, интервал находится в регистре в конце базового блока, но выводится в память в начале последующего блока, необходимо вставить инструкцию перемещения, чтобы сохранить регистр в памяти при обработке этого края потока управления. Бинпакинг второго шанса выполняет анализ потока данных, чтобы минимизировать количество вставленных перемещений.

Попробуем код предыдущего примера обработать с помощью бинпакинга второго шанса. Первые два интервала v1 и v2 получают физические регистры r1 и r2 соответственно. Когда в линейном проходе по всем интервалам

достигается v3, интервал v1 только что достиг дыры в жизни. Поэтому v1 содержится не в активном списке, а в новом списке, называемом неактивным. Этот список содержит все интервалы, которые начинаются до и заканчиваются после текущей позиции, но в данный момент находятся в дырке времени. Физические регистры неактивных интервалов могут быть частично назначены другим интервалам. В примере, интервал v3 получает физический регистр r1 без какого-либо перелива.

Анализ асимптотической временной сложности бинпакинга второго шанса более сложен. В то время как фактический проход по всем интервалам выполняется за линейное время, как в базовом алгоритме, другие части, такие как анализ потока данных, не могут быть выполнены за линейное время. В итоге, общая асимптотическая временная сложность превышает O(n). Однако измерения показывают, что только несколько процентов от общего времени распределения тратится на нелинейные части, поэтому жертвование линейностью не имеет большого влияния. Таким образом, бинпакинг второго шанса почти так же быстр, как базовый алгоритм линейного сканирования. Это хороший компромисс, если важны как время компиляции, так и время выполнения программы.