Распределение регистров методом раскраски графа несовместимости для VLIW-архитектур

Боханко А.С., Дроздов А.Ю., Новиков С.В., Шлыков С.Л.
ИМВС РАН, г. Москва
{ruff|sasha|novikov|shlykov}@mcst.ru

Аннотация

В оптимизирующем компиляторе бесспорно важная роль принадлежит распределителю регистров. Наиболее удачным из известных подходов к распределению регистров остается подход, основанный на раскраске графа несовместимости. Однако же, все известные алгоритмы, основанные на этом подходе, рассчитаны на последовательное выполнение операций, и не пригодны для архитектур с широким командным словом (VLIW-архитектур). В настоящей работе проводится анализ проблем, возникающих при распределении регистров для архитектур подобного рода, и предлагаются методы решения этих проблем.

1. Введение

Одной из наиболее перспективных парадигм выполнения вычислений являются архитектуры с явной поддержкой параллелизма на уровне отдельных инструкций, известные также как архитектуры с широким командным словом.

Согласно [11], архитектурами с широким командным словом называются архитектуры, допускающие одновременное выполнение определенного числа инструкций, закодированных либо как одна большая инструкция, либо как фиксированный набор инструкций с явным указанием параллелизма между отдельными инструкциями. Такой набор инструкций получил название широкой команды. В англоязычных работах архитектуры с широким командным словом носят название VLIW-архитектур (сокращение от Very Long Instruction Word).

Самой известной коммерчески успешной реализацией концепции параллелизма на уровне отдельных инструкций является семейство микропроцессоров Itanium компании Intel ([16]). Также стоит отметить архитектуру E2K, разработанную отечественной компанией Эльбрус ([8]).

Разработчики Itanium расширили концепцию широкого командного слова. Их архитектура позволяет использовать широкие команды не фиксированной, а произвольной длины; впрочем, параллельное выполнение операций в таких командах не гарантируется, что налагает дополнительные ограничения на исполняемый код. Эта расширенная версия широкого командного слова получила название EPIC (Explicitly Parallel Instruction Computing).

Общепризнано, что для VLIW-архитектур оптимизирующий компилятор является ключевым звеном в достижении наивысшей производительности. Больше того, без эффективного компилятора потенциал таких архитектур раскрыть невозможно. В свою очередь, для оптимизирующего компилятора важнейшее значение играет распределение регистров. Это одна из немногих задач, без решения которых компилятор вообще не способен получить корректно работающий код. В настоящий момент наиболее мощным, гибким и эффективным подходом к распределению регистров является подход, основанный на раскраске графа несовместимости.

Самой ранней работой, в которой раскраска графа несовместимости использовалась для решения задачи распределения ограниченного числа ресурсов, следует считать [1]. В ней был заложен теоретический базис, и изложение намеренно абстрагировано от каких-либо свойств и особенностей существовавших в то время вычислительных машин.

Это позволяет указанной работе оставаться актуальной и сегодня, хотя некоторые размышления Ершова имеют лишь теоретический интерес. Например, он предлагает использовать редукцию

для ускорения процесса раскраски графа несовместимости; но анализ применения данной техники на практике показывает, что накладные расходы, необходимые для такой редукции, сводят на нет все ее преимущества, лишь замедляя раскраску.

Первой работой, в которой раскраска графа рассматривалась исключительно с практической точки зрения применения ее в распределителе регистров оптимизирующего компилятора, стала [5], написанная в 1981 году. В следующем году появилась новая, дополненная на основе практического опыта версия той же статьи ([6]). Впоследствии обе статьи Чайтина стали широко известны, и на них принято ссылаться как на первоисточники во всех работах, затрагивающих вопросы распределения регистров. Пальма первенства, однако же, принадлежит Ершову; беда в том, что результаты его исследований стали известны на Западе относительно поздно.

Во всех последующих работах значительных модификаций алгоритма предложено не было. Ряд статей уделял внимание ускорению алгоритма или уменьшению требуемой им памяти ([10], [12]; как правило, результаты оказывались очень скромными, и выдающихся достижений в этой области так и не получилось), еще несколько статей было посвящено обсуждению различных вариантов эвристик, оценивающих стоимость откачки виртуальных регистров в память и выбора регистров для откачки на этапе раскраски графа ([2], [7], [4], [3]).

Хороший обзор всех этих работ представлен в монографии Стивена Мучника ([13]). В ней дано подробное описание как самого алгоритма, так и всех новаций, предложенных за истекшие со времени появления статьи Чайтина годы. Повторим еще раз, что особенных новаций, изменяющих основные принципы алгоритма, не было.

В то время, когда были написаны работы Ершова и Чайтина (конец 70-ых – начало 80-ых годов), вычислительные системы не обладали многими архитектурными механизмами, привычными сегодня. В частности, не было широкого командного слова и поддержки параллелизма на уровне отдельных инструкций. Соответственно, ни в монографии Ершова, ни в статьях Чайтина о влиянии этих механизмов на процесс распределения регистров не сказано ни слова. Но это влияние есть, и оно очень велико. Настолько велико, что алгоритм распределения регистров, не учитывающий его, становится практически бесполезным и даже некорректным. Эта тема остается все еще малоисследованной.

В последующих разделах настоящей работы будет уделено пристальное внимание влиянию широкого командного слова на процесс распределения регистров, основанный на раскраске графа несовместимости. Во втором разделе авторы рассмотрят проблемы, возникающие при пропагации времени жизни регистров – действии, необходимом для построения графа несовместимости. Будут предложены новые алгоритмы обхода операций, корректно работающие для архитектур с поддержкой параллелизма на уровне отдельных инструкций. Далее будут рассмотрены вопросы взаимодействия распределителя регистров и планировщика – еще одной крайне важной компоненты оптимизирующего компилятора. В последнем разделе будет подведен итог работы и сделаны выводы.

2. Построение графа несовместимости

Все архитектуры, поддерживающие параллелизм на уровне отдельных инструкций, можно условно разделить на VLIW- и EPIC-архитектуры.

С точки зрения распределения регистров, основное различие между ними в следующем: в случае VLIW гарантируется, что перед тем, как будут выполнены операции, входящие в широкую команду, все их аргументы будут прочитаны. Это позволяет использовать для аргументов и результатов одни и те же номера физических регистров.

EPIC не гарантирует одновременного выполнения всех операций; поэтому при выполнении программы операции, стоящие в начале широкой команды, могут уже определить свои результаты еще до того, как будут прочитаны аргументы следующих за ними операций. Таким образом, одинаковые номера регистров для некоторого аргумента операции `o1` и результата

операции o2 можно использовать только в том случае, если o1 находится в широкой команде раньше, чем o2. Такое свойство EPIC-архитектур получило название *последовательной семантики (sequential semantics)*. Действительно, при распределении регистров мы должны полагать, что возможно последовательное выполнение всех операций, входящих в широкую команду. Однако, в отличие от действительно последовательных машин, EPIC налагает одно важное ограничение: так как операции могут выполняться и параллельно, то все результаты операций, входящих в одну широкую команду, должны определять разные регистры. Это ограничение справедливо и для VLIW.

Описанные выше свойства заставляют изменить алгоритм пропагации значений внутри линейного участка. Для EPIC-архитектур изменения минимальны: все, что нужно сделать, это построить в графе несовместимости дуги между узлами, соответствующими результатам операций, входящих в одну широкую команду. На рисунке 2.1 предложен алгоритм, выполняющий описанные действия.

```
procedure InterfereBlockOpersReses( block) {
    foreach w_instr ( BlockWInstrs( block) ) {
        foreach oper ( WInstrOpers( w_instr) ) {
            foreach oper2 ( WInstrOpers( w_instr) ) {
                InterfereOpersReses( oper, oper2);
            }
        }
    }
}
```

Рисунок 2.1. Алгоритм построения дополнительных дуг в графе несовместимости для результатов операций

Функция InterfereOpersReses строит в графе несовместимости дуги, выражающие несовместимость между результатами двух операций; ее реализация полностью определяется используемыми структурами данных.

Как легко понять, алгоритмическая сложность предложенного алгоритма равна O(o * o), где o – число операций линейного участка. Однако, к общей сложности алгоритма пропагации времени жизни внутри линейного участка это не добавляет ровным счетом ничего, так как его собственная сложность имеет тот же порядок.

Для VLIW-архитектур изменения алгоритма пропагации более значительны. Во-первых, справедливо все сказанное выше для EPIC. Но кроме того, необходимо учесть, что аргументы операций широкой команды читаются всегда раньше, чем определяются результаты, и не создавать в графе несовместимости лишних дуг.

Требуется изменить порядок обхода операндов операций. Если в традиционном алгоритме вначале обрабатывается результат операции, потом ее аргументы, а затем алгоритм переходит к следующей операции, то теперь нужно вначале обработать результаты всех операций, входящих в широкую команду (в любом порядке), потом аргументы этих операций (опять-таки, в любом порядке), а затем перейти к операциям следующей широкой команды.

На рисунке 2.2 приведен алгоритм такого обхода.

```
/* Операции линейного участка обходятся по широким командам. */
foreach w_instr ( BlockWInstrs( block) ) {
    /* Вначале обрабатываем все результаты операций широкой команды. */
    foreach oper ( WInstrOpers( w_instr) ) {
        ProcessOperRes( oper, in_vector);
    }

    /* Затем обрабатываем все аргументы операций широкой команды. */
    foreach oper ( WInstrOpers( w_instr) ) {
        ProcessOperArgs( oper, in_vector);
    }
}
```

Рисунок 2.2. Алгоритм пропагации внутри линейного участка для VLIW-архитектур

Функций ProcessOperRes и ProcessOperArgs выполняют те же действия, что и в традиционном алгоритме пропагации. А именно, ProcessOperRes строит в графе несовместимости дуги между результатом заданной операции и всеми живыми регистрами, и сбрасывает флаг “жизни” регистра, которому принадлежит результат операции. ProcessOperArgs устанавливает флаг “жизни” тех регистров, которым принадлежат аргументы заданной операции.

Важно то, что обе эти функции изменяют значение in_vector (битового вектора, хранящего информацию о том, какие регистры “живы” в данный момент) и то, в каком порядке вызываются эти функции.

К алгоритмической сложности алгоритма пропагации времени жизни внутри линейного участка, опять-таки, не добавляется ничего. Как и раньше, обходятся операнды всех операций; все что изменилось, это порядок обхода.

Однако же, практически время работы увеличивается: ведь теперь вместо одного обхода операций нужно делать два. Можно обойтись и одним обходом, но в этом случае необходимо использовать два вектора. На рисунке 2.3 приведен алгоритм, позволяющий и в случае VLIW-архитектур ограничиться лишь одним обходом всех операций широкой команды:

```
in_vector2 = NewZeroVector( size( in_vector));

foreach w_instr ( BlockWInstrs( block) ) {
    foreach oper ( WInstrOpers( w_instr) ) {
        ProcessOperRes( oper, in_vector);
        ProcessOperArgs( oper, in_vector2);
    }

    in_vector = bit_or( in_vector, in_vector2);

}
```

Рисунок 2.3. Альтернативный алгоритм пропагации внутри линейного участка для VLIW-архитектур

Функция NewZeroVector создает новый битовый вектор указанного размера, все элементы которого равны нулю. По завершении обхода операций широкой команды два битовых вектора объединяются в один по ИЛИ (функция bit_or).

Основная идея алгоритма заключается в том, что функция ProcessOperArgs работает только с вектором in_vector2 (все элементы которого вначале равны нулю), а функция ProcessOperRes только с in_vector. Таким образом, по окончании обхода операций широкой команды в in_vector2 будут отмечены все узлы графа несовместимости, время жизни которых началось в этой команде. В узле in_vector же будут сброшены флаги “жизни” каких-то узлов, и кроме того, функция ProcessOperRes может создать новые дуги в графе несовместимости. В конце работы над широкой командой результаты вновь объединяются в один in_vector.

Алгоритмическая сложность двух предложенных алгоритмов не отличается; она равна O(o * n), где o – число операндов линейного участка, а n – число узлов в графе несовместимости (то есть суммарное число всех физических и виртуальных регистров). Точно такой же сложностью обладает и “стандартный” алгоритм пропагации, описанный в работе [6].

Какой из двух алгоритмов пропагации для VLIW-архитектур выбрать, определяется исключительно требованиями к оптимизирующему компилятору (например, там где есть ограничение по памяти, больше подойдет первый алгоритм; а там, где требуется строгое разделение программы на функции и максимальное переиспользование этих функций, лучше подходит второй).
В заключение необходимо отметить, что алгоритм пропагации между линейными участками остается без изменений.

3. Взаимодействие распределителя регистров и планировщика

В архитектурах с широким командным словом важная роль принадлежит планировщику. Его задачей является максимальное задействование всех устройств микропроцессора, то есть наибольшее заполнение широкой команды. Для VLIW-архитектур наиболее удачным считается алгоритм планирования, предложенный в работе [9].

Естественно, что всякое агрессивное планирование приводит к возрастанию числа одновременно живущих переменных, что усложняет задачу распределения регистров и может привести к появлению дополнительных операций откачки/подкачки. Более того, чем больше регион работы планировщика, тем вероятнее возрастание времени жизни переменных.

Напомним, что операцией откачки (spill) называется операция, временно сохраняющая значение регистра в памяти. Такие операции необходимы в тех случаях, когда в некоторой точке программы не хватает физических регистров для распределения на них всех “живущих” в этот момент виртуальных. Операция подкачки (fill) – это операция, выполняющая обратное действие – загрузку значения из памяти на регистр. Очевидно, что откачка/подкачка крайне негативно влияет на скорость работы программы.

Распределитель регистров, в свою очередь, приводит к появлению новых зависимостей, так как два ранее независимых виртуальных регистра могут быть распределены на один физический регистр, и сделать, таким образом, соответствующие операции зависимыми друг с другом. Чем больше зависимостей, тем менее оптимальными становятся результаты работы планировщика.

Возникает некоторый “заколдованный круг”, когда от порядка запуска фаз зависят результаты их работы, а идеального порядка нет и быть не может.

Авторами были проведены исследования того, какой порядок работы планировщика и распределителя регистров наиболее эффективен. Результаты этих исследований собраны в таблице 1. Для замеров использовался общепринятый в индустрии и исследовательском сообществе пакет тестов SPEC2000, в котором собраны наиболее типичные пользовательские программы. В него включены как целочисленные, так и вещественные задачи.

Название теста	Эффективность работы распределителя регистров (вначале планирование)	Эффективность работы распределителя регистров (вначале распределение)	Эффективность работы планировщика (вначале планирование)	Эффективность работы планировщика (вначале распределение)
100.twolf	4082	4006	99715	100554
176.gcc	21221	21314	655531	661618
255.vortex	10366	10544	175275	175317
253.perlbmk	8209	8219	229534	230048
181.mcf	423	404	3877	3951
164.gzip	1056	1042	15434	15630
254.gap	10740	10447	250182	253460
177.mesa	13096	12957	174363	188354
179.art	453	466	7971	8135
183.equake	510	397	5390	5461
197.parser	3714	3689	94939	96183
256.bzip2	756	743	12845	13295
188.ammp	3638	3523	53294	55105
186.crafty	1927	1926	62756	63336

Таблица 1. Результаты замеров эффективности работы распределителя регистров и планировщика в зависимости от порядка запуска этих фаз

Под “эффективностью работы планировщика” понималось число широких команд, в которые ему удалось спланировать процедуру. Под “эффективностью работы распределителя регистров” понималось число операций откачки/подкачки, которые ему потребовалось создать для того, чтобы распределить все виртуальные регистры процедуры. Планировщик и распределитель были настроены на архитектуру Эльбрус 3М (максимальная ширина команды 14 (существуют некоторые ограничения по типам планируемых операций), число регистров общего назначения 192, число предикатных регистров 64).

Как видно из таблицы, возросшее после планирования давление на регистры практически не сказалось на эффективности работы распределителя регистров. В то же время, результаты работы планировщика при его запуске после распределения оказались всегда хуже, чем при запуске до распределения. Статическое ухудшение (увеличение числа широких команд) в большинстве случаев не очень велико, однако исследования показывают, что динамическое ухудшение (увеличение числа тактов, необходимых для выполнения задачи) гораздо более существенно. Дело в том, что при измерении статических показателей большую погрешность вносят “обязательные” широкие команды, не зависящие от порядка запуска фаз. Это пролог, эпилог, команды, содержащие вызовы процедур, переходы, не входящие в циклы, и им подобные широкие команды.

Некоторые исследователи (например, в [14]) пропагандируют одновременное планирование и распределение регистров. Но как показывает опыт, приобретенный при разработке оптимизирующего компилятора для вычислительной системы Эльбрус 3М, в современных условиях такой подход нежизнеспособен. Дело в том, что для сегодняшних пользовательских программ характерны большой размер и высокая сложность. Больше того, оптимизирующие преобразования, как правило, приводят к увеличению этой сложности и еще большему запутыванию структуры управления. В то же время, как уже было отмечено, размеры регистровых файлов в современных архитектурах очень велики, а правила использования регистров могут быть достаточно ограничительными. Поэтому для максимально эффективного задействования имеющихся регистровых ресурсов в программах со сложной структурой управления необходим глобальный (в рамках всей процедуры) анализ, который при планировании, работающем в рамках отдельных линейных участков, провести невозможно.

Как правило, при одновременном планировании и распределении регистров используется битовый вектор, в котором отмечаются уже занятые физические регистры. Для нового виртуального регистра выделяется первый свободный в этом векторе физический регистр. Если свободных физических регистров нет, то один из виртуальных регистров, занимающих физический, откачивается в память. Таким образом, в первую очередь распределяются всегда те виртуальные регистры, которые встречаются в линейном участке раньше, что начисто отрицает возможность оптимального распределения регистров во всей программе.

Кроме того, так как планировщик работает локально в рамках одного линейного участка, то на входе этого участка невозможно сказать, какие регистры уже заняты. Поэтому накладывается дополнительное ограничение на локальность (в рамках линейного участка) всех регистров, как виртуальных, так и физических.

В составе оптимизирующего компилятора проекта Эльбрус 3М было реализовано распределение регистров, проводившееся одновременно с планированием. Результаты оказались плачевными – как эффективность работы планировщика, так и эффективность работы распределителя регистров много хуже, чем в случае использования глобального распределителя регистров и любого из двух порядков запуска планирования и распределения.

Таким образом, результаты проведенных нами исследований показывают, что для современных VLIW- и EPIC-архитектур наиболее оптимальным является распределение регистров на уже спланированном коде. То есть вначале должен запускаться планировщик, а за ним следовать распределитель регистров.

Заключение

В настоящей работе мы рассмотрели проблемы, возникающие при распределении регистров, основанном на раскраске графа несовместимости, в архитектурах с широким командным словом.

Нами были предложены алгоритмы, позволяющие корректно и оптимально проводить пропагацию времени жизни и построение графа несовместимости для VLIW- и EPIC-архитектур.

Далее мы подробно изучили вопрос порядка следования и взаимодействия планировщика и распределителя регистров; нами были приведены результаты тестирования, показывающие что при решении практических задач оптимальным порядком является тот, когда распределение регистров проводится после планирования.

Литература

[1] Андрей Ершов, “Введение в теоретическое программирование (беседа о методе)”, Наука, Москва, 1977

[2] P. Briggs, K. D. Cooper, K. Kennedy, L. Torczon, “Coloring heuristics for register allocation”, Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, 1989, pp. 275-284

[3] Preston Briggs, Keith D. Cooper, Linda Torczon, “Improvements to graph coloring register allocation”, ACM Transactions on Programming Languages and Systems, Vol. 16, Issue 3 (May 1994), pp. 428-455

[4] David Callahan, Brian Koblenz, “Register allocation via hierarchical graph coloring”, Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, 1991, pp. 192-203

[5] Gregory Chaitin, Mark Auslander, Ashok Chandra, John Cocke, Martin Hopkins, Peter Markstein, “Register allocation via coloring”, Computer Languages, January 1981, pp. 47-57

[6] Gregory Chaitin. “Register allocation and spilling via graph coloring”, Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, June 1982, pp. 98-105

[7] Fred C. Chow, John L. Hennessy, “The priority-based coloring approach to register allocation”, ACM Transactions on Programming Languages and Systems, Vol. 12, Issue 4 (October 1990), pp. 501-536

[8] Keit Diefendorf, “The russians are coming: supercomputer maker Elbrus seeks to join x86/IA-64 melee”, Microprocessor Report, Vol. 11, Num. 2, Feb. 15, 1999

[9] Joseph A. Fisher, “Trace scheduling: a technique for global microcode compaction”, IEEE Transactions on Computers, Vol. C-30, July 1981, pp. 478-490

[10] Rajiv Gupta, Mary Lou Soffa, Denise Ombres, “Efficient register allocation via coloring using clique separators”, ACM Transactions on Programming Languages and Systems, Vol. 16, Issue 3 (May 1994), pp. 370-386

[11] John L. Hennessy, David A. Patterson, “Computer architecture: a quantitative approach”, Morgan Kauffman, San Francisco, 2003

[12] Christine Makowski, Lori L. Pollock, “Achieving efficient register allocation via parallelism”, Proceedings of the 1995 ACM symposium on Applied computing, 1995, pp. 123-129

[13] Steven S. Muchnick, “Advanced compiler design and implementation”, Morgan Kauffman, San Francisco, 1997, chapter 7.2

[14] Cindy Norris, Lori L. Pollock, “An experimental study of several cooperative register allocation and instruction scheduling strategies”, Proceedings of the 28th annual international symposium on Microarchitecture, 1995, pp. 169-179

[16] Walter Triebel, “Itanium architecture for software developers”, Intel Press, 2000