Прелюдия

Причиной появления данной статьи является устойчивое ощущение автором всеобщей избалованности технологическими достижениями в области микроэлектроники, результатом чего является некоторая леность ума и нежелание применять интеллект там, где можно просто взять молоток побольше.

Компиляторы (и генераторы кода в том числе) принципиально не менялись последние 20 лет. Но если языки программирования за то же время так же принципиально не менялись, то потенциал вычислительных средств увеличился революционно. При этом, как ни странно, и архитектура процессоров за это время сколь-нибудь существенно не изменилась. Микропроцессоры маскируются под самих себя двадцатилетней давности, кодогенераторы делают вид, что верят, все довольны.

Автор пытается задуматься о цене такого согласия и о возможно существующих архитектурных и организационных решениях, оставшихся не найденными только потому, что никому не пришло в голову их искать. В качестве затравки предлагается разобрать работу стекового процессора.

Еще раз об оптимальной генерации кода.

Борис Муратшин (zzeng@mail.ru)

суббота, Август 6, 2005

Содержание

1 Введение **2**

2 Стек **3**

3 Компиляция **4**

4 Промежуточная архитектура **7**
4.1 Pro и contra 7
4.2 Инструкции общего назначения 8
4.3 Арифметические/алгебраические инструкции 8
4.4 Логические операции 9
4.5 Битовые операции 9
4.6 Преобразование типов 9
4.7 Прочее 10
4.8 Примеры 10

5 Генерация кода **14**
5.1 Общие операции 15
5.2 Вызов функций 17
5.3 If . . . Then . . . Else 18
5.4 Циклы 18
5.5 Исключения 18

1 Введение

Назовем нашу мысленную конструкцию "резиновый стек"и попробуем представить ее функционирование исходя из не очень ясной пока целевой функции конечной производительности системы. Правильнее было бы дать название вроде "виртуального стека", но слово "виртуальный"за последние годы подверглось такой девальвации, что совершенно не воспринимается слушателем.

Основной содержательной идеей данной работы является введение промежуточной архитектуры, того самого "резинового стека", главным предназначением которого является поддержание баланса между легкостью компиляции и эффективной генерацией кода. При этом мы не фиксируем целевую архитектуру, а наоборот, пытаемся понять, какой она должна быть исходя из конечной эффективности. Сама по себе идея не нова, тот же GCC использует внутреннее стековое представление, правда, для изоляции от деталей целевой платформы, и это как раз тот случай, когда за общность приходится платить производительностью. Мы же пытаемся провести некоторый "синтез"архитектуры при минимальном количестве ограничений.

2 Стек

Почему стек? Koopman [1] пишет: "From a theoretical viewpoint stacks themselves are important, since stacks are the most basic and natural tool that can be used in processing well structured code ... stack machines are also much more efficient at running certain types of programs than register-based machines, particularly programs which are well modularized. Stack machines also are simpler than other machines, and provide very good computational power using little hardware". Зададимся вопросом, так почему же стековые машины в настоящий момент вытеснены регистровыми процессорами на обочину прогресса? Причины, думается, в следующем:

1. Существуют две "крайние"реализации стековых процессоров: с фиксированным аппаратным стеком и стеком, расположенным в памяти. Обе эти крайности имеют существенные изъяны: в случае стека фиксированного размера, мы имеем проблемы с компилятором и/или операционной системой, которые обязаны думать о том, что произойдет, если стек переполнится или опустеет и как отработать соответствующее прерывание. Если обработка прерывания отсутствует, мы имеем дело с микроконтроллером, обреченным на то, чтобы быть программируемым вручную. При наличии обработчика такого прерывания, аппаратный стек играет роль "окна"к памяти с быстрым доступом, при этом сдвиг этого "окна"является довольно дорогой процедурой. В результате мы имеем непредсказуемое поведение программы, не позволяющее использовать такой процессор, например, в системах реального времени, хотя, с точки зрения Koopman'а, это как раз экологическая ниша подобных процессоров.

2. В случае же стека, размещенного целиком в памяти, мы платим обращением к памяти за практически каждую исполняемую инструкцию. Таким образом, общая производительность системы ограничена сверху пропускной способностью памяти.

3. Сложность процессора перестала быть сдерживающим фактором в развитии.

4. Аппаратно реализованный стек подразумевает строгую последовательность происходящих событий, а, следовательно, и невозможность использовать неявный параллелизм программ. Если суперскалярные процессоры сами распределяют инструкции по конвейерам в соответствии с имеющимися ресурсами, а VLIW процессоры ждут, что эта работа уже выполнена за них компилятором, то для стековых машин попытка найти в коде скрытый параллелизм сталкивается с почти непреодолимыми трудностями. Иными словами, мы снова сталкиваемся с ситуацией, когда технология может дать больше, чем архитектура может себе позволить. Что странно.

5. Программирование для регистровых машин более "технологично". Имея всего лишь несколько регистров общего назначения, "вручную"легко можно создать код, который будет обращаться к памяти лишь тогда, когда этого действительно нельзя избежать. Отметим, что в случае стековой машины достичь этого намного труднее. И, хотя оптимальное распределение временных переменных по регистрам является NP-полной задачей [3], существует несколько недорогих, но действенных эвристик, позволяющих создавать вполне приличный код за разумное время.

В силу указанных причин, наша целевая архитектура является регистровой, суть нововведений находится в алгоритме генерации кода и аппаратной поддержке этого алгоритма.

3 Компиляция

В данном разделе мы кратко остановимся на существующих технологиях компиляции. Эти технологии были разработаны в 60-80е годы, но на настоящий момент принципиально не изменились [3].

Компиляцию условно можно разбить на несколько шагов:

1. *Синтаксический анализ*. На этом этапе осуществляется синтаксически управляемая трансляция, производятся статические проверки. На выходе мы имеем дерево (даг[1]) синтаксического разбора.

2. *Генерация промежуточного кода*. При желании, генерация промежуточного кода может быть объединена с синтаксическим анализом. Тем более, что при использовании в качестве промежуточного кода трехадресных инструкций, данный шаг становится тривиальным ибо "трехадресный код является линеаризованным представлением синтаксического дерева или дага, в котором внутренним узлам графа соответствуют явные имена"[3]. Если присмотреться, трехадресный код есть код для виртуального процессора с бесконечным количеством регистров.

   Рассмотрим этот трюк поподробнее. Хотя, трехадресный код действительно есть лишь способ записи синтаксического дерева и никакой потери информации не происходит, здесь возникает очень важная бифуркация. С

[1] Directed Acyclic Graph

одной стороны, мы имеем дерево и стек как наиболее естественный способ работы с ним. С другой, мы вводим некоторый виртуальный процессор с (если угодно) "резиновым"пулом регистров. Отметим, что явной необходимости вводить новую сущность в виде регистров на данном шаге не существует, это всего лишь констатация того факта, что в силу вышеуказанных причин, целевая машина все равно будет регистровой и поэтому в некотором смысле разумно закрепить этот факт уже сейчас.

3. *Генерация кода*. Результатом данного шага является программа для целевой архитектуры. Здесь мы поподробнее остановимся на последствиях архитектурных решений, принятых при реализации предыдущего шага.

   Поскольку реальное количество регистров ограничено и невелико, на этом этапе мы должны определить, какие временные переменные в каждый момент будут находиться в регистрах, и распределить их по конкретным регистрам. Даже в чистом виде эта задача является NP-полной, кроме того, дело усложняется тем, что обычно существуют разнообразные ограничения по использованию регистров. Тем не менее, для решения данной задачи разработаны приемлемые эвристики. Кроме того, трехадресный (или его эквиваленты) код дает нам формальный аппарат для анализа потоков данных, оптимизации, удаления бесполезного кода, ...

   Предположим, что в качестве промежуточного представления мы выбрали код для стекового процессора. Как нам генерировать оптимальный код для регистровой целевой архитектуры? Кажется весьма привлекательным использовать имеющиеся регистры для вершины стека. Остальную часть стека мы можем расположить в памяти и уповать на то, что она будет эффективно кэшироваться. При этом компилятор в каждый конкретный момент должен помнить, какой регистр занимает какую позицию в стеке, PUSH будет означать, что самый "далекий"регистр выталкивается в память и становится вершиной стека. POP приводит к "подкачке"из памяти в регистр, ADD осуществляется между двумя регистрами на вершине стека ... Здесь сразу возникает ряд "но":

   - Нельзя полагаться на то, что, к примеру, 32 регистров заведомо хватит на вычисление любого выражения. В качестве примера можно привести цепные дроби [2] , которые нельзя упростить алгебраически и которые могут потребовать для своего вычисления стека любой глубины. Справедливости ради отметим, что это случай скорее экзотический и что для вычисления подавляющего большинства выражений требуются не более десятка регистров.
   - А вот случай передачи через стек большого количества аргументов экзотическим не является. А, следовательно, не является редкой ситуация, когда нам заведомо не хватит регистров и для расположения всех вычисленных аргументов и для вычисления следующего. Т.е. в

[2] выражения типа: $[a_0, a_1, a_2, a_3, \ldots] = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \ldots}}}$

этой ситуации любое движение стека приведет к обращению к памяти.

- Часто локальные переменные, вводимые программистом, являются временными, предназначенными для того, чтобы разгрузить текст программы. И если при использовании трехадресного кода, такие переменные наряду с прочими (созданными автоматически) временными переменными участвуют в оптимизации и назначении регистров, то в случае стековой машины, устранение этих переменных вызывает трудности.
- Не стоит рассчитывать на эффективное кэширование вершины стека. В некоторых ситуациях[5] вообще стараются избегать кэширования памяти в силу непредсказуемости поведения программы. Но и обычных условиях, более - менее длинный цикл по массиву данных с большой вероятностью вытеснит стековые данные во всяком случае из кэша первого уровня. То же произойдет и при возврате из процедуры с нетривиальным фреймом данных, что отнюдь не редкость. Переключение же потоков просто не оставляет шансов найти стековые данные в кэше какого бы то ни было уровня.
- Существует определенная проблема с вызовом процедур. Действительно, информацией, о том, в каком порядке регистры расположены на вершине стека, обладает только компилятор. Очевидно, после возврата из процедуры, регистры должны содержать те же данные в том же порядке. И заниматься этим восстановлением должна вызывающая сторона. Разумным решением представляется "выталкивание"всех данных стека из регистров в память и размещение аргументов функции в регистрах (насколько их хватит) в строго определенном порядке. Впрочем, это не сильно отличается от существующей практики, когда после возвращения из процедуры содержимое всех (или почти всех) регистров считается испорченным.
- Аналогично обстоят дела с переходами внутри процедуры. Если мы можем попасть в некоторую метку из разных мест, значит мы либо должны восстанавливать порядок регистров, либо "выталкивать"все значения из регистров в память. Впрочем, и в существующей практике метка считается началом базового блока [3] с соответствующими последствиями для оптимизатора.

Тем не менее, мы имеем простой и регулярный алгоритм генерации кода для стекового внутреннего представления и регистровой целевой архитектуры. Причем, помимо очевидных недостатков этот алгоритм имеет и немаловажные преимущества, наиболее значимыми из которых являются: органически бережное отношение к количеству использованных регистров и отсутствие необходимости решать NP - полную задачу по распределению этих самых регистров.

[3] последовательность инструкций с единственными входом и выходом потока управления, без возможности ветвления [3]

4 Промежуточная архитектура

4.1 Pro и contra

Сейчас мы попробуем спроектировать стековую промежуточную архитектуру, максимально использующую собственные преимущества и избавленную от недостатков (или хотя бы минимизирующую их). Попробуем еще раз взглянуть на недостатки, присущие стековым процессорам:

1. Проблема ограниченного аппаратного стека. Ограничения на глубину стека делают затруднительной регулярную генерацию кода, поэтому у нас не будет подобных ограничений.

2. Проблема излишне частых обращений к памяти. Поскольку мы не имеем ограничений на глубину стека, этот стек в конечном счете расположен в памяти, и обойти этот факт невозможно. Однако, для борьбы с частыми обращениями к дорогому ресурсу существуют хорошо зарекомендовавшие себя методы: буферизация, предварительная подкачка, асинхронная запись, ... В любом случае это не проблема промежуточной архитектуры и мы переносим ее решение на более поздний этап генерации кода.

3. Проблема недостаточной сложности стековых процессоров отсутствует в силу отсутствия физической реализации стекового процессора.

4. Проблема строгой последовательности выполнения инструкций стековым процессором отсутствует опять же в силу отсутствия физической реализации стекового процессора. А так как целевой архитектурой является регистровый процессор, мы можем либо положиться на то, что неявный параллелизм будет устранен суперскалярным ядром, либо попытаться сделать его (параллелизм) явным, если, конечно, целевой процессор дает нам такие возможности.

5. Проблема технологичности программирования не является самостоятельной. Если мы считаем, что в состоянии справиться с массовым доступом к памяти, она перестает быть проблемой и становится конкурентным преимуществом.

Итак, приступим. Каноническая стек - машина [1] подразумевает наличие TOS [4] регистра, что, по большому счету является всего лишь деталью реализации, специфическим способом обращения с элементом на вершине стека. Логически, наша действительно каноническая (пуританская, если угодно) стековая машина оперирует только со стеком.

Какие типы данных поддерживает наша машина? Все элементарные типы данных, доступные пользователю C/C++. Здесь мы будем рассматривать лишь INT(32) и DOUBLE(64). Для понимания этого достаточно, а расширение при необходимости представляется самоочевидным.

Какого размера элемент стека? Не важно. Достаточного для того, чтобы при необходимости вместить элемент любого типа.

[4] Top Of Stack

4.2 Инструкции общего назначения

- IMDPUSH_(INT32|DBL) VAL - на вершину стека заносится элемент, являющийся непосредственным аргументом инструкции
- VARPUSH ARG - на вершину стека заносится элемент, содержащий адрес переменной - аргумента
- EVAL_(INT32|DBL) - разыменование переменной, адрес переменной заменяется ее значением
- POP_INT32 - удаление элемента с вершины стека
- DEL_(INT32|DBL) IDX - удаление произвольного элемента стека с соответствующим индексом (один из элементов, делающих наш стек "резиновым")
- LIFT_(INT32|DBL) IDX SHIFT - подъем произвольного элемента стека с соответствующим индексом на SHIFT позиций вверх (. . .)
- SINK_(INT32|DBL) IDX SHIFT - погружение произвольного элемента стека с соответствующим индексом на SHIFT позиций вниз (. . .)
- COPY_(INT32|DBL) SHIFT - элемента стека с соответствующим индексом заносится на вершину стека (. . .)
- ARRPUSH VAL - на вершине стека находится адрес массива или структуры, этот адрес заменяется указателем на элемент массива или структуры, находящийся от изначального на VAL байт
- ASSIGN_(INT32|DBL) - первый элемент с вершины стека заносится по адресу, хранящемуся во втором элементе стека, после чего второй элемент удаляется

4.3 Арифметические/алгебраические инструкции

- ADD_(INT32|DBL) - суммируются значения пары верхних элементов стека, вместо них в стек добавляется получившаяся сумма
- SUB_(INT32|DBL) - берется разница пары верхних элементов стека, вместо них в стек добавляется получившаяся разность
- MUL_(INT32|DBL) - берется произведение пары верхних элементов стека, вместо них в стек добавляется получившееся произведение
- DIV_(INT32|DBL) - пара верхних элементов стека делится, вместо них в стек добавляется получившийся результат деления
- MOD_(INT32|DBL) - вычисляется остаток от деления пары верхних элементов стека, вместо них в стек добавляется получившийся остаток
- UMINUS_(INT32|DBL) - унарный минус, значение на вершине стека заменяется на свое отрицание

4.4 Логические операции

- AND - пара целочисленных значений на вершине стека замещается результатом применения логического AND
- OR - пара целочисленных значений на вершине стека замещается результатом применения логического OR
- NOT - логическое отрицание, значение на вершине стека заменяется своим отрицанием

4.5 Битовые операции

- BAND - пара целочисленных значений на вершине стека замещается результатом применения битового AND
- BOR - пара целочисленных значений на вершине стека замещается результатом применения битового OR
- BNOT - у элемента на вершине стека переворачиваются все биты
- BXOR - пара целочисленных значений на вершине стека замещается результатом применения XOR

4.6 Преобразование типов

- INT32_TO_DBL IDX - элемент стека с соответствующим индексом меняет тип с целочисленного на double
- DBL_TO_INT32 IDX - действие, обратное предыдущей инструкции

Управление потоком

- IF ADDR - если целочисленное значение на вершине стека ноль - переход по адресу ADDR
- JUMP ADDR - безусловный переход
- CALL ADDR - вызов процедуры, адрес следующей инструкции при этом заносится в стек управления, параметры уже находятся в стеке данных
- RET - возврат из процедуры, происходит переход по адресу из вершины стека управления, который при этом укорачивается

4.7 Прочее

Приведенных выше инструкций достаточно для кодирования любой программы. При этом реализованная аппаратно стековая машина будет регулярно выполнять заведомо бесполезные действия. Например, присвоение $a = 1$; будет закодировано как:

```
VARPUSH          A
IMDPUSH_INT32    1
ASSIGN_INT32
POP_INT32
```

Причем, последовательность ASSIGN_INT32,POP_INT32 появляется очень часто и есть соблазн заменить ее чем-нибудь вроде FINASSIGN_INT32. Аналогичным целям служат инструкции SWAP (меняет местами два элемента), DUP (дублирует элемент), OVER (заносит в вершину стека второй элемент) [2].

4.8 Примеры

Код для $a = 1$; мы уже приводили, попробуем что-нибудь посложнее.

$a = b * d + b * e + c * d + c * e$;, если не предпринимать никаких усилий ни по алгебраической нормализации, ни по оптимизации, превращается в

Инструкция	Arg	Содержимое стека после ее выполнения
VARPUSH	A	(&A)
VARPUSH	B	(&A,&B)
EVAL_INT32		(&A,B)
VARPUSH	D	(&A,B&D)
EVAL_INT32		(&A,B,D)
MUL_INT32		(&A,B*D)
VARPUSH	B	(&A,B*D,&B)
EVAL_INT32		(&A,B*D,B)
VARPUSH	E	(&A,B*D,B,&E)
EVAL_INT32		(&A,B*D,B,E)
MUL_INT32		(&A,B*D,B*E)
ADD_INT32		(&A,B*D+B*E)
VARPUSH	C	(&A,B*D+B*E,&C) [5]
EVAL_INT32		(&A,B*D+B*E,C)
VARPUSH	D	(&A,B*D+B*E,C,&D)
EVAL_INT32		(&A,B*D+B*E,C,D)
MUL_INT32		(&A,B*D+B*E,C*D)
ADD_INT32		(&A,B*D+B*E+C*D)
VARPUSH	C	(&A,B*D+B*E+C*D,&C)
EVAL_INT32		(&A,B*D+B*E+C*D,C)
VARPUSH	E	(&A,B*D+B*E+C*D,C,&E)
EVAL_INT32		(&A,B*D+B*E+C*D,C,E)
MUL_INT32		(&A,B*D+B*E+C*D,C*E)
ADD_INT32		(&A,B*D+B*E+C*D+C*E)
ASSIGN_INT32		(A)
POP_INT32		()

Что бросается в глаза при взгляде на этот код? Несмотря на то, что переменные b, c, d, e не меняются, каждое обращение к ним влечет за собой загрузку значения из памяти. Попробуем после каждой загрузки переменной сохранять копию значения.

[5] здесь & обозначает ссылку на переменную

Инструкция	Arg	Arg	Содержимое стека после ее выполнения
VARPUSH	A		(&A)
VARPUSH	B		(&A,&B)
EVAL_INT32			(&A,B)
COPY_INT32	0		(&A,B,B)
VARPUSH	D		(&A,B,B,&D)
EVAL_INT32			(&A,B,B,D)
COPY_INT32	0		(&A,B,B,D,D)
SINK_INT32	0	2	(&A,B,D,B,D)
MUL_INT32			(&A,B,D,B*D)
COPY_INT32	2		(&A,B,D,B*D,B)
VARPUSH	E		(&A,B,D,B*D,B,&E)
EVAL_INT32			(&A,B,D,B*D,B,E)
COPY_INT32	0		(&A,B,D,B*D,B,E,E)
SINK_INT32	0	3	(&A,B,D,E,B*D,B,E)
MUL_INT32			(&A,B,D,E,B*D,B*E)
ADD_INT32			(&A,B,D,E,B*D+B*E)
VARPUSH	C		(&A,B,D,E,B*D+B*E,&C)
EVAL_INT32			(&A,B,D,E,B*D+B*E,C)
COPY_INT32	0		(&A,B,D,E,B*D+B*E,C,C)
SINK_INT32	0	3	(&A,B,D,E,C,B*D+B*E,C)
COPY_INT32	4		(&A,B,D,E,C,B*D+B*E,C,D)
MUL_INT32			(&A,B,D,E,C,B*D+B*E,C*D)
ADD_INT32			(&A,B,D,E,C,B*D+B*E+C*D)
COPY_INT32	1		(&A,B,D,E,C,B*D+B*E+C*D,C)
COPY_INT32	3		(&A,B,D,E,C,B*D+B*E+C*D,C,E)
MUL_INT32			(&A,B,D,E,C,B*D+B*E+C*D,C*E)
ADD_INT32			(&A,B,D,E,C,B*D+B*E+C*D+C*E)
SINK_INT32	0	4	(&A,B*D+B*E+C*D+C*E,B,D,E,C)
POP_INT32			(&A,B*D+B*E+C*D+C*E,B,D,E)
POP_INT32			(&A,B*D+B*E+C*D+C*E,B,D)
POP_INT32			(&A,B*D+B*E+C*D+C*E,B)
POP_INT32			(&A,B*D+B*E+C*D+C*E)
ASSIGN_INT32			(A)
POP_INT32			()

Алгоритм получения этого кода таков:

1. Все выражения, которые могут быть повторно использованы (в данном случае - значения переменных), мы дублируем и помещаем копию за границы текущего выражения. В приведенном примере они размещаются в стеке сразу после адреса переменной A.

2. Когда нам нужно значение предварительно вычисленного подвыражения, мы копируем его на вершину стека.

3. После того, как текущее выражение вычислено, все повторно - использованные подвыражения удаляются из стека

Необходимо отметить следующие моменты:

1. Мы действительно свели к минимуму обращения к памяти

2. Максимальная использованная нами глубина стека - 8, что относительно много.

3. Если бы мы удаляли временные копии подвыражений сразу за последним использованием. средняя глубина использования стека уменьшилась бы примерно на единичку.

4. Если бы инструкция ASSIGN брала адрес присваиваемой переменной с вершины стека, а не с предпоследнего элемента, это уменьшило бы глубину использования стека еще на единичку.

Повторно используемые значения могут быть вынесены за границы текущего базового блока и использованы в пределах всего базового блока, попробуем продемонстрировать это подробнее в следующем примере:

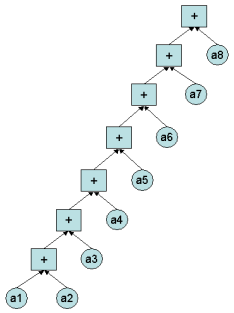
c = a + b;
a = c - 1;
b = a + 5;

при этом получаем псевдокод:

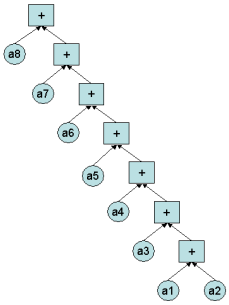
Инструкция	Arg	Arg	Содержимое стека после ее выполнения
VARPUSH	C		(&C)
VARPUSH	A		(&C,&A)
COPY_INT32	2		(&A,&C,&A)
EVAL_INT32			(&A,&C,A)
VARPUSH	B		(&A,&C,A,&B)
COPY_INT32	3		(&B,&A,&C,A,&b)
EVAL_INT32			(&B,&A,&C,A,B)
ADD_INT32			(&B,&A,&C,A+B)
COPY_INT32	2		(&B,&A,A+B,&C,A+B)
ASSIGN_INT32			(&B,&A,A+B)
IMDPUSH_INT32	1		(&B,&A,A+B,1)
SUB_INT32			(&B,&A,A+B-1)
ASSIGN_INT32			(&B,A')
IMDPUSH_INT32	5		(&B,&A,A',5)
ADD_INT32			(&B,&A,&C,A'+5)
ASSIGN_INT32			(A'+5)
POP_INT32			()

Максимальная глубина использования стека здесь - 5.

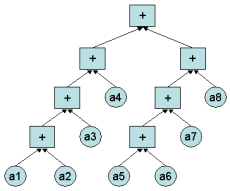
Теперь поговорим немного об упрощении выражения. До сих пор мы не прибегали к данному приему. Впрочем, задача эта весьма нетривиальная и представляется разумным оставить ее на ответственности конечного программиста (кроме немногочисленных простых случаев). Рассмотрим это на примере выражения $a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$. По сути, это n-арный оператор сложения, но поскольку фактический оператор сложения бинарен, мы имеем множество представлений этого выражения. Ниже представлены некоторые варианты: Фиг.1 есть (т.е. программист может явно задать как) $(((((((a_1+a_2)+a_3)+a_4)+a_5)+a_6)+a_7)+a_8)$, Фиг.2 представляет собой $(a_1+(a_2+(a_3+(a_4+(a_5+(a_6+(a_7+(a_8)))))))$, Фиг.3 - это $((((a_1+a_2)+a_3)+a_4)+(((a_5+a_6)+a_7)+a_8))$, а Фиг.4 - $((((a_1+a_2)+(a_3+a_4))+(a_5+a_6))+(a_7+a_8))$.



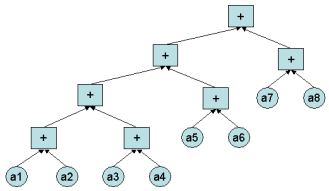
фиг. 1



фиг. 2



фиг. 3



фиг. 4

Если мы введем целевую функцию пригодности представления для наших целей, то сможем определить оптимальное и, следовательно, алгоритм нормализации выражений.

Допустим, мы оптимизируем максимально использованную глубину стека. Тогда Фиг.1 даст нам значение - 2, Фиг.2 - 7, Фиг.3 - 3, и Фиг.4 - также 3. Пользуясь коммутативностью и ассоциативностью оператора +, мы за линейное время можем получить оптимальный код для однородных выражений. Для генерации кода для разнородных выражений можно воспользоваться алгоритмом динамического программирования [6].

Если же целевой функцией является максимальная производительность, задача немного усложняется. Описанный алгоритм предполагает строгую последовательность вычислений и минимальный внутренний параллелизм. Если же мы знаем о наличии у целевой архитектуры способностей параллельно исполнять инструкции, описанный алгоритм не позволит эти способности использовать. В самом деле, в случае Фиг.1, для того, чтобы просуммировать a3, мы обязаны дождаться результата a1 + a2. Выходом может быть пожертвовать оптимальностью использования стека. Если мы взглянем на Фиг.4, то увидим, что глубина использования стека увеличилась на единицу, но при этом суперскалярное ядро процессора будет в состоянии обнаружить независимые суммиро-

[6] Алгоритм динамического программирования разделяет задачу генерации оптимального кода для выражения $E = E_1\ op\ E_2$ на подзадачи генерации оптимального кода для подвыражений за чем следует код вычисления оператора op [3]

вания, то же самое можно сказать и про последующую стадию компиляции для ILP[7] архитектуры. Нетрудно показать, что Фиг.4 является в этом смысле оптимальной.

5 Генерация кода

Вот и настало время превратить наш промежуточный код в нечто реальное. В качестве целевой архитектуры возьмем типичный регистровый процессор с типичным набором трехадресных инструкций.

Итак, у нас есть некоторое количество регистров общего назначения. Будем считать их 32-разрядными, хотя, по большому счету это не важно. Так же как не важно их количество, которое есть лишь параметр в нашей модели и подлежит оптимизации.

В качестве демонстрации мы используем архитектуру OpenRISC 1000[6], впрочем, этот выбор в значительной мере случаен и не обусловлен какими либо специфическими особенностями данной архитектуры.

Верхушку стека мы будем размещать в регистрах. То, что по тем или иным причинам не смогло поместиться в регистрах, помещается в область сверхбыстрой памяти с прямой адресацией, работающей как кольцевой буфер. То, что не смогло разместиться и там, размещается в регулярной памяти. Промежуточная память, как уже указывалось, служит нам для трех основных целей:

1. Буферизация
2. Загрузка по предположению
3. Асинхронная запись.

Для работы с этой специализированной памятью мы вводим дополнительный набор инструкций (с namespace'ом lm):

- **lm.push Ri**. Вносит значение из регистра Ri. Если память переполнена, эта инструкция вызовет сохранение некоторого числа элементов из конца буфера в оперативную память. При этом на период записи блокируются кольцевые указатели, но не блокируется доступ к данным. В общем случае данная инструкция не блокирует процессор, т.е. выполняется за 1 такт.
- **lm.pop Ri**. Обратная операция, удаляет элемент из начала буфера и заносит его в указанный регистр. При недозаполненности буфера может вызвать загрузку нескольких элементов данных из оперативной памяти. В этом случае блокируются кольцевые указатели, но не блокируется доступ к данным. В общем случае, эта инструкция также работает асинхронно, выполняется за 1 такт.
- **lm.copy Ri, I**. Осуществляет доступ к данным, заносит элемент с глубины I в регистр Ri. Выполняется за 1 такт.

[7] Instruction Level Parallelism

- **lm.land Ri**. Содержимое буфера целиком выгружается в оперативную память. Выполняется асинхронно.
- **lm.uplift Ri**. Содержимое буфера целиком загружается из оперативной памяти. Выполняется асинхронно. Может выполняться одновременно с выгрузкой старого содержимого по другому адресу.
- **lm.pushall**. Выталкивает все регистры в кольцевой буфер. Как вариант можно предположить аргумент - битовую маску регистров, подлежащих выталкиванию.
- **lm.popall**. Подгружает все регистры из кольцевого буфера. Как вариант можно предположить аргумент - битовую маску регистров, подлежащих загрузке.

Привязка этого буфера к адресному пространству осуществляется через специальный регистр операционной системой при инициализации thread'а. При смене контекста операционная система должна выгрузить старые данные, загрузить новые, привязать буфер к новому адресному пространству. Все это можно сделать в фоновом режиме без явных потерь, по крайней мере, один способ [4] известен.

5.1 Общие операции

Основные операции, которые нам надо исследовать, это занесение в стек и извлечение из стека. Рассмотрим их на примерах.

1. Стек пуст, кодируем инструкцию "VARPUSH x". Нам следует выбрать любой регистр и занести в него адрес переменной x. Т.е. все сводится к

   l.addi R3, R2, I

   где I - сдвиг x от указателя на фрейм, а R2 используется как этот самый указатель, R0 зарезервирован, R1 - указывает на верхушку стека.

2. В стеке один элемент, кодируем инструкцию "EVAL_INT32". Нам следует загрузить значение из указателя, равного указателю на фрейм + сдвиг. Получаем

   l.ld R4, 0(R3)

   Здесь мы взяли R4 для хранения результата, если R3 (адрес переменной, который мы разместили в этом регистре в предыдущем примере) нам больше не нужен, мы (т.е. компилятор) можем считать этот регистр свободным.

3. Глубина стека меньше количества выделенных под него регистров, кодируем инструкцию "POP_INT32". Здесь ничего делать не надо, компилятор всего лишь должен запомнить, что регистр, который был на вершине стека, теперь свободен.

4. Число элементов в стеке равно NR[8]-1, кодируем инструкцию "VARPUSH x". Отметим, что нам потребуется один регулярный ("блуждающий") регистр под временные значения. Неявно мы это использовали в предыдущем примере. Т.к. мы не можем использовать регистр при загрузке слова и как указатель и как приемник данных, необходим еще один регистр, который становится ненужным как только инструкция закончилась. Таким образом, количество элементов стека, которые мы можем разместить в регистрах, на единицу меньше числа регистров, которые мы под это выделили. В данном случае наша задача усложняется т.к. помимо занесения адреса мы должны вытолкнуть элемент из регистра в кольцевой буфер.

   l.addi R3, R2, I
   lm.push R8

   В данном случае мы использовали "блуждающий"регистр R3 для хранения адреса переменной, освободили регистр R8 (например), который компилятор нашел на глубине стека NR-1. Теперь регистр R8, в свою очередь, стал "блуждающим". Отметим, что приведенные инструкции независимы и суперскалярным ядром могут быть выполнены параллельно.

5. Число элементов в стеке больше NR, кодируем инструкцию "POP_INT32". Регистр, который был найден компилятором на вершине стека (пусть R5) теперь окажется на глубине NR и в него мы поместим значение с вершины кольцевого буфера.

   lm.pop R5

6. Число элементов в стеке больше NR, кодируем ADD_INT32.

   l.add R5, R8, R7
   lm.pop R7

   Здесь R5 - был "блуждающим"регистром, R7 находился на вершине стека, R8 - ячейкой ниже. После указанных инструкций компилятор будет считать R5 находящимся на вершине стека, R8 станет "блуждающим", а R7 окажется на глубине NR-1 и будет содержать элемент с вершины кольцевого буфера.

7. Число элементов в стеке больше NR, кодируем COPY_INT32 2.

   lm.push R7
   l.add R7, R0, R5

   Здесь R0 - аппаратный 0, R7 компилятор нашел на глубине NR-1 и выталкивает в кольцевой буфер. Потом в этот регистр копируются данные из регистра на вершине стека и далее компилятор будет считать R7 находящимся на глубине 2. Стоит заметить, что максимальная глубина, на которую стоит выполнять псевдо-инструкцию COPY_INT32, это NR. В

[8] число выделенных под стек регистров

этом случае мы просто выполняем *lm.push Rx*. Копирование на большую глубину слишком дорого и проще перевычислить то значение, которое мы пытаемся спасти.

8. Число элементов в стеке больше NR, кодируем SINK_INT32 0, 2.

   lm.push R7
   l.add R7, R0, R5

   Забавно, но в данном случае код этой псевдо-инструкции совпадает с предыдущим примером. Разница в том, что сейчас регистр R7 окажется на вершине стека.

9. Число элементов в стеке больше NR, кодируем SINK_INT32 с глубины больше NR.

   lm.push R7
   lm.copy R7, I

   Здесь R7, как водится, был на глубине NR-1, а окажется на верху.

5.2 Вызов функций

На первый взгляд - ничего особенного, заносим аргументы в стек и вызываем функцию. Проблема в том, что функция, которую мы вызываем, может понятия не иметь, в каком порядке наши регистры составляют вершину стека. Разумным способом добиться совместимости представляется предопределенный порядок использования регистров. Например, регистр R3 должен содержать первый аргумент функции (или первое слово первого аргумента), R4 - второй аргумент (...), ... Если нам не хватит регистров для передачи всех аргументов, в дело вступает кольцевой буфер.

Если же еще остались элементы стека в регистрах, их придется вытолкнуть в кольцевой буфер. При этом, если выталкивать надо небольшое количество данных, это можно сделать явно, а если, например, у функции вообще нет аргументов, выгоднее воспользоваться командой **lm.pushall** и вытолкнуть их все. Порядок регистров в данном случае не важен т.к. далее (после возврата из функции) мы их восстановим через **lm.popall**.

Возврат из процедуры должен восстановить состояние стека. В C/C++ принято, что вызывающая сторона вычищает аргументы из стека после возврата из функции. Плюс к этому необходимо еще обработать значение возврата функции.

Поскольку мы договорились о предопределенном порядке расположения аргументов в регистрах, значение возврата (при компиляции **return**) мы можем расположить так же (пользуясь теми же соглашениями), как если бы это был еще один аргумент. Тогда вызывающая сторона сможет сохранить его копию и поместить в стек после выполнения **lm.popall** или проведет аналогичные действия, если регистры спасались явно.

5.3 If . . . Then . . . Else

- Пусть мы имеем блок с одной точкой входа и одной точкой выхода. В этой ситуации любая из веток кода может сделать с регистрами все, что ей заблагорассудится. На момент объединения веток, состояние стека, разумеется, обязано остаться неизменным, но элементы на его вершине могут быть распределены по регистрами в разном порядке. Например, одна ветка не использует стек, а вторая успела полностью вытолкнуть содержимое регистров в кольцевой буфер и вернуть его обратно. Самое простое решение этой проблемы - компилятор должен гарантировать для всех веток один порядок использования регистров - именно тот, в каком они содержали вершину стека на момент ветвления.

- Оператор **return** внутри ветвления. При возврате из процедуры, мы должны вернуть стек в том же состоянии, в каком его получили. Но эта проблема уже не относится к собственно обработке ветвления.

- Операторы **continue**, **break** внутри ветвления. Обрабатываются аналогично. Для нашего ветвления это внешняя проблема и решаться она должна в том месте, к которому относится **continue** или **break**.

5.4 Циклы

Циклы сильно похожи на ветвления. По выходу из цикла, состояние стека не должно измениться относительно входа в цикл. Обеспечить это не сложно, достаточно сохранять порядок выделения/освобождения регистров при вычислениях внутри итерации. Как поступить с операторами **continue** и **break**? Компилятор должен лишь откатить состояние стека, пользуясь единым порядком регистров.

5.5 Исключения

Исключения и связанная с ними раскрутка стека - одна из самых приятных методик в программировании на C++.

"Исключения согласуются с динамической блочной структурой программы. . . . При возбуждении исключений динамическая блочная структура "разматывается"в процессе восстановления от наименьшего блока, содержащего ошибочный оператор, до первого динамически объемлющего блока, содержащего обработчик соответствующего исключения"[7]. И в ADA и традиционно в C++[8] исключения рассматриваются буквально, как исключительные ситуации, ошибки, после которых надо аккуратно восстановиться. Это выражается в том, что обработка исключения весьма дорога.

Если бы не эта дороговизна, исключения можно было бы трактовать шире - как обобщенный и расширенный способ передачи управления - **continue**, **break**, **goto**, **longjmp** и **return** в одном флаконе. Попробуем разобраться, что именно влияет на стоимость обработки исключений.

Существует два подхода к реализации их обработки. Первый демонстрируется компилятором GCC, его можно охарактеризовать словами "пусть неудачник платит"[9]. Компилятор старается минимизировать издержки, связанные с обработкой исключений в тех случаях, когда они не происходят. При этом вокруг стековых объектов с нетривиальными деструкторами и мест, где потенциально возникают исключения, размножается служебный код и древовидные служебные данные. Раскрутить стек, пользуясь такой конструкцией, просто, но очень дорого[9].

Компилятор Microsoft Visual C++ использует другую стратегию - по немногу платят все и за всё[10], а именно:

- Для каждой функции, которая потенциально может выбрасывать исключения, компилятор создает в качестве локальной переменной структуру EXCEPTION_REGISTRATION, эта структура регистрируется в **FS:[0]** и с помощью указателя на предыдущую аналогичную структуру образует стек.
- Компилятор создает таблицу структур - по элементу для каждого **try**-блока в функции. Каждый **try** блок имеет номер начала и конца (вложенный блок имеет вложенный интервал), соответствующий некоторому номеру состояния, за актуальностью которого сам компилятор и следит. Нетрудно заметить, что таким несколько экзотическим образом реализуется стек **try**-блоков.
- На каждый **try**-блок заводится таблица catch блоков.
- На каждый тип исключения заводится таблица **type_info** всех базовых классов в иерархии данного исключения.
- Для каждой функции создается **unwind** таблица, каждый элемент которой содержит указатель на функцию, освобождающую ресурс и номер предыдущего элемента. В одной таблице может быть расположено несколько цепочек в зависимости от областей видимости объектов с деструкторами. В момент исключения, по номеру текущего состояния, который мы уже упоминали, используя его как индекс unwind таблицы, можно пройти нужную цепочку и вызвать все необходимые деструкторы. Разумеется, это еще одна эмуляция работы стека из адресов процедур.

Фактически, мы имеем три вспомогательных стека в добавок к основному стеку данных. Поскольку физический стек в архитектуре IA32 только один, компилятор вынужден паковать все четыре стека в одном пространстве и использовать изощренные способы навигации по ним. При этом, вспомогательные стеки (стеки управления) работают синхронно и по сути являются проявлениями одного и того же. Ну не может область видимости переменной выходить за рамки **try**-блока, а **try**-блок за пределы функции, так же как не могут **try**-блоки или области видимости пересекаться лишь частично.

[9] © А.Артюшин

Поэтому, в нашей целевой архитектуре мы введем два стека: один для данных и один для адресов, указывающих на исполняемый код (стек управления). Ситуацию, когда семантически разные значения находятся вперемешку, мы сочтем недопустимой.

Как мы будем различать элементы разных типов в стеке управления? По значению. Все эти элементы суть указатели на объекты, расположением которых компилятор может управлять. Так, например, он может выравнивать их в памяти по границе 4 или 8 байт. А это значит, что младшие 2 или 3 бита значения стека управления могут быть использованы для идентификации типа элемента.

Итак:

- перед вызовом процедуры, в стек данных заносятся параметры
- при вызове процедуры, в стек управления заносится адрес возврата
- при начале блока видимости, в стеке данных отводится место под локальные переменные
- каждый раз, когда конструируется стековый объект с виртуальной таблицей, в стек управления заносится адрес этого объекта. Поскольку такой объект обязан иметь виртуальный деструктор и компилятор может расположить его с предопределенным индексом, при раскрутке стека управления удалить этот объект не составит труда.
- каждый раз, когда конструируется стековый объект без виртуального деструктора, в стек управления заносится адрес кода, удаляющего этот объект исходя из его адреса в текущем фрейме. Альтернативным решением может быть принудительная виртуализация всех нетривиальных деструкторов
- для каждого **catch** обработчика в стек управления заносится указатель на структуру, содержащую **type_info** этого обработчика и указатель на код, обрабатывающий данное исключение
- при выходе из блока видимости, по очереди вызываются и удаляются из стека управления элементы, соответствующие объектам данного блока, после этого из стека данных удаляются локальные переменные.
- при выходе из **try**-блока, из стека управления удаляются все записи, соответствующие catch-обработчикам данного блока
- при выходе из функции, из стека данных удаляется текущий фрейм, происходит возврат по адресу, занесенному в стек управления, после чего этот адрес из стека управления удаляется
- после возврата из функции, из стека данных удаляются параметры
- для каждого оператора **throw** создается статический список из **type_info** всей иерархии классов типа исключения. Конечно, при компиляции без

RTTI[10], вместо списка мы имеем один элемент (и не с **type_info**, а, скорее, с некоторым внутренним номером типа).

- в случае возникновения исключения, сверяясь с этим списком, мы удаляем элементы из стека управления до тех пор, пока не найдем указатель на **catch**-структуру подходящего типа. При этом по ходу дела мы вызываем все встретившиеся деструкторы. Когда подходящий **try/catch** блок найден, вы устанавливаем в нужный фрейм указатель стека данных, выполняем соответствующий восстановительный код и отдаем управление.

В результате, мы имеем простые и кристально-прозрачные алгоритм компиляции и механизм поддержки исполнения исключений. Как результат - компактный код и минимум издержек при исполнении. Ничего лишнего, создаются только те структуры данных, без которых никак не обойтись, и исполняется только тот код, который в любом случае должен был быть выполнен.

При этом аппаратная реализация стека управления желательна, но не обязательна. Компилятор вполне может эмулировать этот стек с помощью пары регистров, физически разместить его в том же сегменте, что и стек данных, но пустить навстречу (например) с другого конца.

Список литературы

[1] Philip Koopman Jr., "Stack Computers: the new wave", "Ellis Horwood", 1989 2, 4.1

[2] Philip Koopman Jr., "A Preliminary Exploration of Optimized Stack Code Generation": 1992 ROCHESTER FORTH CONFERENCE 4.7

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman , "Compilers: Principles, techniques and Tools", "Addison-Wesley", 1986; ISBN 0-201-10088-6. 3, 2, 3, 6

[4] B. Muratshin, A. Artiushin: The persistent cache approach for multitasking and SMP systems. Novosibirsk, Jul 2002, Russian Patent N2,238,584. 5

[5] Kirk, David Brian: Computer system with private and shared partitions in cache; US Patent N5,875,464. 2

[6] OpenRISC 1000 Architecture Manual (http://www.opencores.org) 5

[7] I.C.Pyle, The ADA - programming language. A guide for programmers, Prentice Hall Int. 1981 5.5

[8] Steph Mineart, ANSI ISO C++ Professional Programmers Handbook, Macmillan Computer Publishing, 1999 5.5

[10] Run Time Type Info

[9] G++ internals - Exception Handling,
(http://theoryx5.uwinnipeg.ca/gnu/gcc/gxxint_13.html) 9

[10] The Code Project - How a C++ compiler implements exception handling -
C++ / MFC, (http://www.codeproject.com/cpp/exceptionhandler.asp)

9