ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ АВТОМАТИЧЕСКОЙ ОПТИМИЗАЦИИ ПРОГРАММ

Л.Е. Брусенцов

Л.Е. Брусенцов
Институт Автоматики и Электрометрии СО РАН, ЗАО «Интел А/О»
e-mail: leonid.brusencov@intel.com

Аннотация.

Сегодня большое внимание уделяется энергоэффективности электронных устройств, особенно это касается процессоров как основы большинства из них. Такая характеристика напрямую зависит от качества бинарного кода, произведённого компилятором. Компиляторы – это многогранные, сложно-настраиваемые инструменты, которые часто могут выполнять свою работу лучше. Итеративные подходы поиска позволяют автоматически делать их подстройку под определённый программный продукт, однако работают очень долго и дают плохо предсказуемый результат. В данной работе предлагается метод грубой оценки возможности автоматической оптимизации.

Введение

При разработке сложных программных продуктов приходится жертвовать оптимальностью конечного кода в пользу упрощения всего процесса их создания. Всё большие надежды возлагаются на оптимизирующие компиляторы, особенно это связано с распространением и постоянным обновлением многоядерных архитектур микропроцессоров. Компьютерные технологии проникли во все аспекты нашей жизни, производительность устройств очень важна, и может быть существенно улучшена с помощью оптимизации заложенных программ.

Исследования показывают, что многомерное дискретное пространство производительности кода, получаемого с помощью различных значений опций оптимизации компилятора, не имеет какой-либо выраженной структуры, а представляет скорее произвольный набор пиков. Это объясняет то, что алгоритм случайного выбора и сложный генетический дают сходные результаты поиска за одинаковое количество итераций. В связи с этим в данной работе предлагается ввести некоторую метрику для исходных кодов программ и определять стратегию поиска на основе уже известных статистических данных.

В области оптимизации приложений широко известна техника сбора динамических характеристик кода приложения во время исполнения, особенно полезная для определения часто исполняющихся участков кода. Это позволяет сосредоточить оптимизацию в наиболее выгодных блоках программы, увеличив производительность в целом. Такой функциональностью оснащаются даже некоторые современные компиляторы.

Статические данные представляют собой сжатую информацию об алгоритмической составляющей программы, которые могут быть получены непосредственно из её внутренних представлений в компиляторе. При этом можно получить как некоторое постоянное высокоуровневое описание, так и низкоуровневый результат конкретной компиляции на последней стадии работы – генерации объектного файла. В частности, это позволяет в некоторой степени оценить влияние выбранной стратегии оптимизации в компиляторе.

Большое влияние на время исполнения кода оказывает исполняющее устройство, причём это не только процессорное ядро, но и разного уровня память, шины доступа и прочее. Более того, существенное значение играет программная среда исполнения, что включает в себя операционную систему как посредника всех ресурсов и используемый приложением набор библиотек.

К сожалению, данным списком все факторы влияния не исчерпываются, поскольку заметное влияние способно оказать и внешнее окружение устройства исполнения (от температуры и влажности воздуха до скачков амплитуды и фазы питания). Ввиду невозможности учёта и замера данных факторов предполагается их исключение путём повторных запусков приложения на каждой итерации и вычисления средних величин.

Таким образом, в работе предлагается вести необходимую статистику на метрическом множестве троек $\langle A,S,H\rangle$, где A – множество характеристик кода, S – множество характеристик программной платформы, а H – множество характеристик исполняющего устройства.

Обзор литературы

Итеративные оптимизации – очень популярный подход в ответ на всё более усложняющиеся архитектуры процессоров. Полный поиск в пространстве значений опций может дать заметный (в 2.65 раз на задаче перемножении матриц) прирост производительности по сравнению со статическими моделями [1], существуют доказательства, почему именно такие подходы приносят прирост на современных архитектурах [2].

Появились публикации, в которых объясняется, как данный подход можно использовать на практике, например, для улучшения параметров оптимизации в библиотеках [3] или для лучшего определения статической модели выбора параметров оптимизаций.

Существует огромное количество статей, посвящённых данной области исследования, можно выделить несколько смежных направлений деятельности. Это jit-компиляция, использование нескольких скомпилированных вариантов функций, и многое другое. Были разработаны целые системы работы с трансформациями (банки знаний), позволяющие осуществлять оптимизирующие и даже распараллеливающие преобразования программ в непроцедурном виде [4]. Но нигде нет законченного решения, позволяющего получать наилучший код для всех ситуаций, как нет и методик оценки каким он должен быть.

Совершенствуются и отдельные трансформации, например алгоритмы GCSE (global common subexpressions elimination) и PRE (partial redundancy elimination), обладая широкими возможностями по адаптации, позволяет сократить среднее время работы тестов Spec95 на 10% [5]. Такие величины считаются очень значительными.

С некоторыми другими подходами можно ознакомиться в [6]. Данная работа основана на результатах работы инструмента IFDO, описанного в [7].

Анализ исходного кода программы

Для анализа используются данные об исходном коде программы, получаемые с помощью эмулятора процессора PIN 2.6 и модифицированного компилятора Intel® Compiler 12.1, а также результаты профилировки с помощью Intel® PTU 3.2 на процессоре Intel® Core 2 Duo. На первой итерации работы IFDO оптимизируемое приложение строится со

значениями всех опций по умолчанию, при этом компилятор дополнительно определяет список содержащихся в нём функций и его статические характеристики. Затем получившийся исполняемый файл запускается через эмулятор PIN для накопления статистики исполнения каждой из его функций, а именно – количества использованных инструкций и данных об использовании памяти и кэша процессора (исполнение при этом замедляется в 23 раза). Затем производится профилировка того же файла с помощью Intel® PTU 3.2 и все полученные данные сохраняются в базе данных.

Низкоуровневые характеристики кода удобно собирать с помощью известного эмулятора процессоров PIN. Потенциально такой эмулятор способен совершить исчерпывающий анализ цикла исполнения программы, однако это бы занимало слишком много времени, а в результате получался огромный массив информации (десятки-сотни гигабайт разрозненных данных). К сожалению, нет известных методов обобщения данных для выявления краткого, но достаточного характеристического описания. Данные для каждой функции приложения собираются эмулятором по-отдельности.

Алгоритмические характеристики проще всего получить из внутреннего представления программы в компиляторе на ранних стадиях его работы (то есть до внесения оптимизационных изменений). Обычно и в нашем случае это представление является деревом ограниченного количества видов предложений (statements) и выражений (expressions), представляющих собой достаточный набор программных алгоритмических единиц, на которых строятся все программы и языки программирования. С помощью таких единиц определяется алгоритмическая структура каждой функции.

Поскольку в наши намерения не входило создание искусственного интеллекта, мы ограничились лишь определением некоторого набора алгоритмических шаблонов и их выявлением в каждой функции программы (то есть каждая функция представляется как множество шаблонов). Определённо это неидеальный, очень ограниченный способ оценки, не учитывающий полностью вкрапления функций друг в друга, протяжения констант, векторизации и прочих оптимизаций. Зато он быстро работает и при хорошем выборе шаблонов позволяет находить сходные по структуре блоки кода.

Удивительным образом оказывается, что производительность приложения сильно варьируется в зависимости от операционной системы, в которой оно исполняется. К сожалению, имеется большое количество и иных факторов, таких как настройки операционной системы, версии динамических библиотек, разрядность кода (наиболее распространены 32-х и 64-х битные) и прочее. Помимо этого также следует учитывать версию компилятора и статических библиотек, которые он использовал для сборки тестового приложения.

С помощью средств операционной системы возможно собрать сведения об исполняющем устройстве, в частности, даже определить все используемые динамические библиотеки и их версии. Из внутренних настроек операционной системы Windows учитывалась лишь одна – интервал переключения процессов планировщиком задач.

Также учитывалось количество свободной оперативной памяти, список запущенных процессов (в частности – служб), время работы ОС без перезагрузки и прочее. Возможно, следует в будущем учесть степень фрагментированности дисковой подсистемы (если она не SSD) и/или добавить запуск дефрагментации, проверки диска и перезагрузки перед началом

работы. Исполняемой задаче повышался приоритет до высокого уровня (high) и назначалась маска используемых ядер процессора (affinity mask) для уменьшения флуктуаций.

Характеристики исполняющего устройства можно получать автоматически только с помощью сервисов операционной системы, однако не всё может быть получено автоматически (например, количество портов процессора, латентности инструкций). В таких случаях приходится выписывать недостающее в таблицы из технической документации, индексирую по идентификаторам устройств.

На данном этапе учитывались только следующие устройства: процессор, оперативная память и северный мост (в нашем случае содержит контроллер памяти). В будущих версиях предполагается также учитывать скорость дисковой подсистемы, лучшем средством измерения которой по праву считается запуск набора специфических тестов (хотя она и описывается некоторыми технологическими величинами и нормами, имеет место большая флуктуация реальных показателей).

В нашем случае требуется некоторая комбинированная метрика из всех имеющихся характеристик. Действительно, наличие и свойства циклов приложения имеют смысл только в связи с размером кэша и прочими свойствами процессора; существуют и другие многочисленные связи.

Поэтому, из всех имеющихся характеристик строится многомерное пространство, где в роли каждого измерения выступает одна из характеристик, и имеется одна выделенная ось – потенциал оптимизации при прочих заданных значениях. В реализации для каждой характеристики хранится распределение потенциала в зависимости от его значений таким образом, что общий потенциал для всех заданных характеристик (в точке пространства) вычисляется как произведение потенциалов каждой отдельности оси пространства. Такая структура данных несколько громоздка, зато гибкая и позволяет рассматривать всё пространство в разрезах (иначе оно просто непредставимо для человека).

С целью отразить тот факт, что для большинства характеристик близкие значения приводят к близким результатам, изменение уровня потенциала оси производилось не точечно, но в некоторой области вокруг точки (сейчас с диаметром 1/3 от всего диапазона значений). Таким образом, например, при переходе на новую версию компилятора мы не начинаем накапливать всю статистику сначала, а подправляем её в соответствии с полученными для него данными.

Статистический анализ возможностей автоматической оптимизации

Анализ возможностей автоматической оптимизации заданного приложения совершается на основе статистических данных об оптимизации заданным компилятором различного вида исходных кодов программ. В данной работе для экспериментов использовалось подмножество тестов из набора SPEC CPU2000. Для начала в режиме обучения запускалась утилита итеративного поиска наилучших опций оптимизации, которая заодно собирала характеристические данные об исходных кодах, получаемые с помощью компилятора, профилировщика VTune и эмулятора PTU.

Затем все полученные данные использовались для уточнения многомерной статистической матрицы потенциалов значений опций компилятора, описанной ранее, а также таблицы достигаемых результатов оптимизации для кодов программ с всевозможными характеристиками. Первая затем используется для ускорения поиска с другими

приложениями, а вторая для грубой оценки потенциально достижимых результатов работы поиска.

Для анализа возможностей автоматической оптимизации требуется по набору характеристик исходного кода после первой итерации работы IFDO найти в базе данных результаты оптимизации наиболее сходных ему оптимизированных ранее приложений. На данном этапе для этого использовалась простая декартова метрика на множестве обобщённых характеристик (например, таких как среднее количество инструкций перехода). Однако, даже с помощью неё удаётся с ощутимой погрешностью предсказывать результаты работы компилятора. Наибольшую проблему в данном случае представляет размер с точки зрения исполнения исследуемого кода (один и тот же цикл с разным количеством итераций исполнения ведёт себя по-разному ввиду специфики современных процессоров).

Метод грубой оценки границы оптимизации

Предлагается следующий способ грубой оценки границы оптимизации для произвольного кода на основе полученных данных (конечно, при этом не берётся в рассмотрение алгоритмическая составляющая). По всем полученным после анализа данным, описанным в предыдущих разделах, оценивается возможность уменьшения/замены тех или иных инструкций в новом оптимизируемом приложении. Затем, с помощью статистической вероятностной модели исполняющего процессора, а также данных профилировки, «исполняется» обобщённый код функций нового приложения и предсказывается время их исполнения.

Модель процессора учитывает некоторые специфические особенности представляемого процессора, например, что инструкций перехода исполнения могут приводить к сбросу конвейера исполнения и потере 14 тактов, а инструкций, работающие с памятью, могут в случае промаха кэша исполняться десятки тактов. В данном случае используется некоторое (пока что статистически заданное) вероятностное распределение для предугадывания времени исполнения каждой из таких инструкций.

В конце концов, на основе данных о времени исполнения каждой функции от профилировщика VTune, вычисляется финальное потенциальное время исполнения приложения. Это число считается грубой границей возможной автоматической оптимизации приложения с помощью данного компилятора для данной архитектуры процессора. Во всяком случае, такая оценка оказывается заметно точнее, чем простая сумма времён исполнения каждой инструкции по-отдельности без учёта задержек и промахов вообще.

Результаты

Конечно, не учитывается большое количество различных факторов, таких как наличие разных портов исполнения инструкций или даже прерываний операционной системы. Однако, даже такая оценка часто оказывается полезной, чтобы отсечь заведомо бесперспективные приложения. Данная методика очень сильно привязана к количеству и качеству собранных ранее данных об оптимизации приложений, а поскольку полный перебор в данном случае невозможен, требуется хороший алгоритм обучения.

Описанная методика тестировалась на двух тестах SPEC CPU2000, для этого сначала совершался прогон около 9000 итераций на одном из них в режиме обучения, затем прогон из 9000 итераций без обучения, и рассматривались результаты алгоритмов предсказания.

Уменьшение времени работы для каждой из функции по-отдельности в среднем удалось предсказать лишь на 40% от достигнутых результатов. В основном это связано с недостатками алгоритма отыскания сходных по характеристикам кодов программ. Зато в лучший найденный набор опций вошло около 70% от числа предугаданных как потенциальных. Соответственно, такой результат вполне может претендовать на существенное сокращение количества необходимых итераций поиска.

Заключение

Как и ожидалось, результаты оказались скромными, однако ввиду большого количества времени, необходимого для совершения итераций поиска наилучших значений опций (а это могут быть годы), даже отдельные проценты ускорения очень важны.

В дальнейшем планируется совершить обучение на половине всех тестов набора SPEC CPU2000 и применить новый алгоритм для оставшейся половины тестов, чтобы более детально определить его эффективность, найти слабые места. Продолжится работа и по изучению эффективности измеряемых характеристик кода, возможно, будут добавлены новые. Более того, в процессе разработки находится новый алгоритм для режима обучения, навеянный анализом уже полученных результатов.

ЛИТЕРАТУРА

[1] Bodin F., Kisuki T., Knijnenburg P., O'Boyle M., and Rohou E. Iterative compilation in a non-linear optimization space. In Proc. ACM Workshop on Profile and Feedback Directed Compilation, 1998, Organized in conjunction with PACT98.

[2] Cooper K., Subramanian D., and Torczon L. Adaptive optimizing compilers for the 21st century. J. of Supercomputing, 32(1), 2002.

[3] Bilmes J., Asanovic K., Chin C., and Demmel J. Optimizing matrix multiply using PHiPAC: A portable, high- performance, ANSI C coding methodology. In Proc. ICS, pages 340-347, 1997.

[4] Князева М. А., Плохих С. А. Концепция системы управления специализированного банка знаний о преобразованиях программ. Информационные технологии, №5, стр. 36-40, 2009.

[5] Филиппов А. Н. Метод нумерации значений и использование его результатов при оптимизации программ. Информационные технологии, №4, стр. 43-49, 2009.

[6] Брусенцов Л. Автоматическая оптимизация при компиляции, Открытые системы, №02, 2011.

[7] Chirtsov A., Brusencov L., Cherny I., Grebenkin S., Ermolaev S. Maximizing Intel® Compiler Performance Using Iterative Feedback Directed Optimization. SECR 2008.