Maximizing Intel® Compiler Performance Using Iterative Feedback Directed Optimization

Artem Chirtsov
Intel Corporation
artem.s.chirtsov@intel.com

Leonid Brusencov
Intel Corporation
leonid.brusencov@intel.com

Ilya Cherny
Intel Corporation
ilya.s.cherny@intel.com

Sergey Grebenkin
Intel Corporation
sergey.s.grebenkin@intel.com

Sergey Ermolaev
Intel Corporation
sergey.n.ermolaev@itnel.com

Abstract

Iterative search of compiler optimization parameters is a popular method to increase performance of compiled code relatively to default compilation. Most of the existing papers apply optimization parameters to the whole application, while our paper describes results got by the specially modified compiler capable to parameterize each function independently. Six well-known search algorithms were implemented and used to search the best optimization parameters in the space of all possible combinations of parameters values. Standard SPEC CPU2000 benchmarks were compiled using internal version of Intel® Compiler 10.1. Applications were executed on Intel® Core 2™ with Windows 2003 OS platforms. Six compiler performance options having the most impact on performance were selected.*

As a result we got the maximum performance increase of 11% for 187.facere benchmark and 4% performance increase of total SPEC CPU2000 execution time. Search algorithms comparison shows that most algorithms get very close results varying within 3% from each other. Some algorithms could get their best results after 10-40 iterations while others could get the same results only after 100 iterations which was the limit of a number of iterations.

***Keywords**: iterative, compiler, optimization, feedback directed, parameters, options, performance*

Максимизация Производительности Компилятора Интел в Режиме Итеративной Оптимизации с Обратной Связью

Artem Chirtsov
Intel Corporation
artem.s.chirtsov@intel.com

Leonid Brusencov
Intel Corporation
leonid.brusencov@intel.com

Ilya Cherny
Intel Corporation
ilya.s.cherny@intel.com

Sergey Grebenkin
Intel Corporation
sergey.s.grebenkin@intel.com

Sergey Ermolaev
Intel Corporation
sergey.n.ermolaev@itnel.com

Abstract

Очень популярной техникой для улучшения производительности скомилированного кода относительно стандартных настроек компилятора сегодня стал итеративный перебор параметров оптимизаций компилятора. Большинство подходов используют для параметризации всё приложения целиком, в данной же статье описывается работа модифицированного компилятора, с которым ведётся параллельный поиск наилучшего набора параметров оптимизаций для каждой функции по-отдельности. Для самого поиска мы выбрали и реализовали шесть наиболее известных алгоритмов и сравнили результаты их работы. Для измерения производительности мы использовали стандартный набор тестов SPEC CPU2000, компьютеры на базе процессоров Intel® Core 2™, Microsoft Windows 2003 и модифицированный компилятор Intel® Compiler версии 10.1. Мы выбрали шесть опций компилятора, влияющие на производительность получаемого кода наибольшим образом.*

В результате мы получили максимальный прирост производительности на отдельном тесте 11%, а на всей сюите в сумме – 4%. Сравнение алгоритмов показало, что различные алгоритмы добиваются в конечном счёте очень близких результатов, в пределах 3% друг от друга, но некоторым для этого достаточно было только 10-40 итераций, тогда как другие добивались сходного результата ближе к 95 итерациям (мы выставляли ограничение на 100 итераций максимум).

***Keywords**: итеративное компилирование, оптимизация, обратная связь, итеративный поиск, производительность, параметры оптимизаций, опции компилятора..*

1. Введение

Чтобы добиться максимальной производительности пользовательских приложений, разработчики компиляторов постоянно увеличивают количество применяемых оптимизаций и изменяют эвристики алгоритмов. Однако, большинство оптимизаций не дают универсальной выгоды для всех приложений и всех сред исполнения (операционная система, вычислительное устройство и т.д.). К тому же, достаточно трудно оценить эффект конкретной оптимизации на финальный код из-за сложной зависимости применяемых оптимизаций друг от друга. Подобные проблемы можно избежать, используя подход итеративного компилирования, то есть метод перебора всех возможных вариантов применяемых оптимизаций.

В данный момент чтобы скомпилировать продукт оптимальным образом, что приходится осуществлять вручную, требуется огромное количество времени, нужно знать и помнить большое количество различных параметров компилятора, выборочно их использовать, затем самостоятельно получать результат и сравнивать его с результатами при других наборах параметров. Мало того, требуется быть знакомым со специализированными программами, измеряющими время исполнения блоков кода, чтобы уметь корректно оценивать результат оптимизации.

Однако, если просто автоматизировать этот процесс, подобная система неизбежно оказывается слишком требовательной ко времени поиска из-за необходимости постоянного перекомпилирования и запуска программы. Но есть возможность сужения пространства поиска наборов опций оптимизации, используя специальные алгоритмы поиска.

В нашей работе используется разбиение программы на отдельные функции, но в дальнейшем будет реализовано разбиение и на более мелкие блоки, например на отдельные циклы. Это позволяет получить дополнительный прирост производительности, которого было невозможно добиться для всей программы целиком.

Для сужения пространства поиска были реализованы шесть различных известных алгоритмов поиска и проведен анализ их работы.

Чем может оказаться полезным наш подход? Скомпилированный оптимальным образом код для трудоёмких вычислений или обработки большого количества данных может позволить получить требуемый результат за гораздо меньшее время, или же повысить качество вычисления с помощью более сложного алгоритма, но за то же время. Максимально возможная оптимизация также очень важна для встраиваемых (embedded) систем.

2. Связанные работы

Сегодня итеративные оптимизации - это очень популярный подход в ответ на всё время усложняющиеся архитектуры процессоров. Так, например, в работе [1] демонстрируется, что полный поиск в пространстве параметров оптимизаций может дать заметный (в статье в 2.65 раз на примере задачи перемножения матриц) прирост производительности в сравнении со статическими моделями, а в [2] дополнительно доказывается почему именно такие работы приносят прирост производительности на современных архитектурах.

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 [4].

Существует проект MILEPOST GCC, описанный в [5], в котором применяется подход машинного самообучения. После нескольких недель накопления статистики удалось добиться 11% уменьшения времени исполнения на тестах MiBench на платформах x86 и IA64.

В [6] описывается подход исследования уровней оптимизации компилятора для автоматического их конструирования, чтобы они представляли оптимальный компромисс между несколькими оценивающими функциями, такими как время исполнения, компилирования, размер кода и другие. Такой компромисс определяется как уровень оптимизации, при котором на всех оценивающих функциях достигается максимум. При этом один уровень доминирует над другим, если он достигает лучшей оценки хотя бы на одной из оценивающих функции, и не хуже на остальных. Для оценки результатов использовалась hyper-volume (HV) метрика, описанная в [7], набор тестов SPEC CPU, процессор Intel Pentium 4 и компилятор GCC. Были получены уровни оптимизации значительно лучшие, чем стандартные -Os, -O1, -O2 и -O3; при оптимизации всех тестов целиком удалось получить до 30% прироста по HV-метрике.

В статье [8] утверждается, что подход, который используется в большинстве итеративных поисков, когда поиск производится на одних и тех же входных данных, демонстрирует лишь потенциал, но не как программа будет работать в реальных условиях. Эта проблема решается подбором большого количества различных входных данных. В самой

статье использовался набор из 20 наборов данных для каждого теста из 26 в MiBench сюите. Эксперименты производились на кластере из 16 AMD Athlon 64 3700+ с помощью PathScale EKIPath Compiler 2.3.1, специально настроенного для AMD процессоров. Было получено, что хотя характеристики программы, такие как производительность, или энергопотребление, могут сильно различаться при использовании нескольких оптимизаций при компилировании, чаще всего возможно найти такой компромиссный оптимальный набор параметров оптимизаций, который бы работал на всех входных данных одинаково. Причём отставание от скомпилированной программы с оптимальными оптимизациями для конкретного набора данных не превышает всего лишь 5%.

В области поиска оптимального набора оптимизаций существует направление, которое называют мультиверсионностью или клонированием [9, 10, 11]. Оно основано на создании нескольких скомпилированных вариантов функций, специализированных на определённые параметры вызова, что позволяет оптимизировать их более агрессивно. При этом ветвление происходит либо статически, либо динамически, либо используя накопленную статистику (например, в [12] в компилятор GCC внедряется машинное самообучение). Такой подход используется как некоторое расширение для компилятора, но не покрывает все случаи различных входных данных в программу.

В работе [13] описывание подход сходный работе JIT (just-in-time) компиляторов. Принцип работы заключается в интерпретации кода для динамического выявления наиболее часто исполняемых мест программы, с последующей их оптимизацией и трансляцией в машинный код. Причём оптимизация ведётся с учётом известных входных данных, то есть часть переменных используются как константы, что добавляет эффективности.

В работе [14] динамически вставляются инструкции упреждающего чтения из памяти, основываясь на счётчиках процессора. В работе [15] используются высокоуровневые оптимизации кода во время исполнения либо в параллельном процессе, либо вообще на удалённой машине. Главная особенность данной работы – она позволяет итеративно искать и применять динамические оптимизации. Пользователю предлагается специальный язык, в котором можно указывать множество эвристик, которые будут применены к определенным участкам программы; затем полученные варианты кода динамически исследуются при запуске, и выбирается лучшее решение. Кроме этого система позволяет генерировать новые варианты кода, основываясь на опыте оптимизации других фрагментов кода.

В работе [16] в исследуемой программе выделяеются фазы исполнения со сходной производительностью, существование которых показывается в [17, 18]. Таким образом, на каждой фазе исполнения можно использовать различные оптимизации, то есть в одном только запуске пробовать сразу несколько решений.

3. Описание реализованных в проекте алгоритмов поиска

Рассмотрим выбраныне нами алгоритмы поиска максимума производительности в пространстве всевозможных комбинаций параметров оптимизаций.

3.1. Алгоритм полного перебора

Алгоритм полного перебора (Exhaustive Search algorithm) использует каждый из допустимых наборов опций компиляции (то есть строит полное пространство решений), в связи с этим найденный минимум целевой функции всегда является глобальным. Однако вычислительная сложность данного алгоритма равна $O(m^n)$, где n – число опций компиляции, m – число возможных значений опций. Такая сложность считается неприемлемой, поскольку время поиска растёт экспоненциально с ростом числа опций [19], поэтому часто на практике предпочтение отдаётся различным модификациям. В данной работе был использован алгоритм полного перебора с приоритетом: каждой опции и её значению на основе экспертной оценки был приписан некоторый вес, в зависимости от него в процессе формирования очередного тестового набора те или иные опции получали более высокий приоритет. В результате при завершении итераций задолго до окончания полного перебора уже будут опробованы наиболее значимые комбинации опций и их параметров.

3.2. Алгоритм группового исключения

Идея алгоритма группового исключения (Batch Elimination Algorithm) заключается в том, чтобы определить опции, которые способствуют увеличению времени исполнения тестируемой программы, и не использовать их в дальнейшем поиске оптимального набора опций. Реализация данного алгоритм достигает хорошей производительности, если влияние опций друг на друга незначительно.

Для определения эффекта использования какой-либо опции используется Относительная

Процентная Доля Улучшения (Relative Improvement Percentage, далее RIP), которая показывает относительную разницу времени исполнения тестируемой программы, скомпилированной в двух вариантах: с выключенной опцией и со всеми включёнными опциями. Если полученная величина отрицательна, данный алгоритм исключает из набора исследуемую опцию, получая, таким образом, оптимальный набор. Вычислительная сложность алгоритма составляет $O(n)$ [20].

3.3. Алгоритм итеративного исключения

В отличие от алгоритма группового исключения, алгоритм итеративного исключения (Iterative Elimination Algorithm) последовательно исключает из стартового набора опции, которые увеличивают время исполнения тестируемой программы. Стартовый набор для данного алгоритма ничем не отличается от стартового набора для алгоритма группового исключения. Обозначим через B_i набор опций B, из которого исключили опцию F_i, тогда после того как для всех наборов B_i значения RIP посчитаны, алгоритм создаёт новый стартовый набор $B^{(1)}$, исключая из него опцию, имеющую RIP с наименьшим отрицательным значением. После чего снова считают RIP для всех наборов $B_i^{(1)}$. Итеративный процесс продолжается до тех пор, пока не будут исчерпаны все опции с отрицательным значением RIP. Вычислительная сложность данного алгоритма составляет $O(n^2)$ [20].

3.4. Алгоритм комбинированного исключения

Алгоритм комбинированного исключения (Combined Elimination Algorithm) совмещает в себе идеи алгоритмов группового и итеративного исключения. Если опции компиляции слабо влияют друг на друга, данный алгоритм исключает опции с отрицательным значением RIP, как алгоритм группового исключения, в противном случае он исключает их в течение итеративного процесса, как алгоритм итеративного исключения. Сложность данного алгоритма равна $O(n^2)$ [21].

3.5. Генетический алгоритм

Генетический алгоритм (Genetic Algorithm) позволяет найти удовлетворительное решение к аналитически неразрешимым проблемам через последовательный подбор и комбинирование искомых параметров с использованием механизмов, напоминающих биологическую эволюцию, в процессе которой выживают наиболее приспособленные особи, а наименее приспособленные погибают.

Алгоритм начинает свою работу с формирования начальной популяции – конечного набора допустимых решений задачи. Эти решения выбираются случайным образом. На каждом шаге эволюции с помощью вероятностного оператора селекции выбираются два решения – родители. Оператор скрещивания по выбранным решениям строит новое, которое подвергается небольшим случайным модификациям – мутациям. Затем решение добавляется в популяцию, а решение с наименьшим значением целевой функции удаляется из популяции [22].

3.6. Алгоритм статистического выбора

Алгоритм статистического выбора (Statistical Selection Algorithm) использует статистику вывода (Inferential Statistics). Вводятся экспериментальная и начальная гипотезы, которые противоречат друг другу. Для проверки гипотез определяют контрольную и экспериментальную группы для каждой из опций, но поскольку для их проверки требуется значительное число запусков тестируемой программы с различным наборами опций, создаётся только одна контрольная и одна экспериментальная группа, в которой значения опций распределены случайным образом с использованием ортогональных массивов, к которым применяется тест Манна-Уитни для расчёта влияния той или иной опции на время программы.

Чтобы определить, влияет ли некоторая опция на время исполнения тестируемой программы, информация о времени исполнения распределяется на две группы – экспериментальную и контрольную, затем, используя информацию экспериментальной группы, вычисляется тестовая статистика, и на её основе при помощи вероятностной функции определяется, попадает ли опция с данным значением в оптимальный набор [23].

4. Методы тестирования

Для тестирования был выбран набор стандартных тестов производительности процессора и памяти SPEC CPU2000 с reference данными, компилятор Intel® Compiler 10.1, компьютеры с процессорами Intel® Core 2™ с частотой 2.40 ГГц и оперативной памятью 2 Гб, и

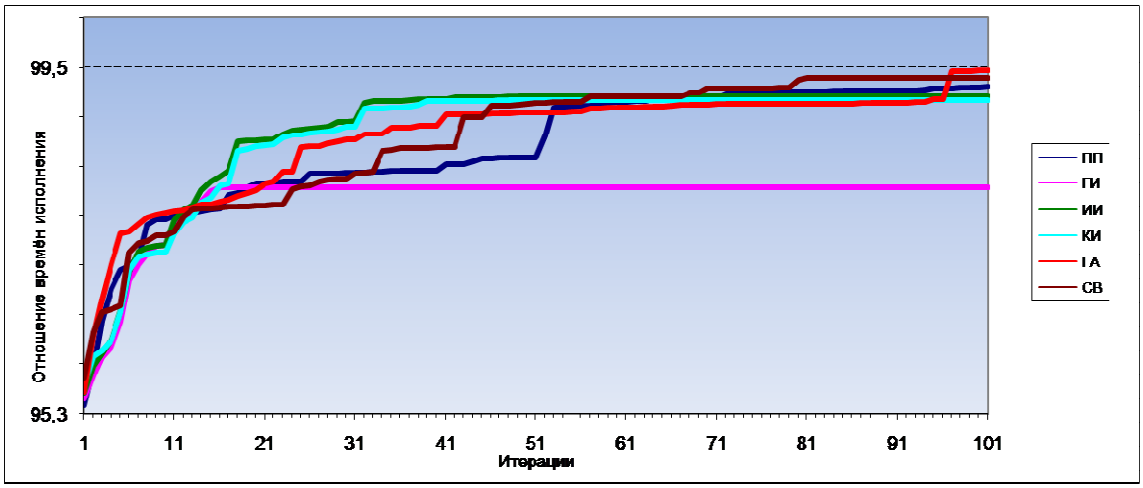

Рисунок 1. График зависимости от итерации суммарного времени исполнения функций всех тестов для каждого из алгоритмов. Показано процентное соотношение минимального времени к текущему.

набор из шести недокументированных опций компилятора, отвечающих за векторизацию (vectorize), слияние циклов (fusion), разбиение циклов (distribution), разворачивание циклов (unroll, unroll and jam), разбиение на блоки (blocking) и инструкции упреждающего чтения из памяти (prefetch). Для измерения времени работы отдельных функций программмы использовался встроенный в компилятор профилировщик, который инструментировал код.

Каждый алгоритм запускался для каждого отдельного теста с ограничением в 100 на максимальное количество итераций. Затем полученные данные анализировались и строились сравнительные графики.

Тестирование проводилось с помощью специально разработанного приложения, производящего автоматически построение и запуск тестов с сохранением данных о результатах в собственную базу данных. Для сравнения было взято шесть описанных выше алгоритмов: алгоритм полного перебора с приоритетом (ПП), алгоритм группового исключения (ГИ), алгоритм итеративного исключения (ИИ), алгоритм комбинированного исключения (КИ), генетический алгоритм (ГА), алгоритм статистического выбора (СВ).

5. Сравнительный анализ

Рассмотрим график зависимости суммарного времени исполнения всех функций всех тестов от номера итерации (Рисунок 1). Было исследовано 164 функций (всего функций 215), входящих в состав различных тестов SPEC CPU2000 (всего 26 тестов). Анализировались только те функции, минимальное время исполнения которых составляло более 1% от минимального времени исполнения данного приложения для всех алгоритмов, чтобы уменьшить уровень шума.

График показывает процентное соотношение суммы минимальных времён исполнения всех функций к сумме времён всех функций рекордных для данной итерации данного алгоритма. Таким образом, график демонстрирует насколько быстро каждый из алгоритмов поиска приближается к максимуму производительности в пространстве всех комбинаций параметров оптимизаций.

Как видно из графика трудно выделить явного лидера, поскольку результаты лидеров отличаются лишь на 0,4%. Наилучшие результаты показали генетический алгоритм, алгоритм статистического выбора и полного перебора (относительное уменьшение времени составило 3,90%, 3,87% и 3,65% соответственно).

Очень близкие друг к другу результаты разных алгоритмов объясняются усреднением по большому количеству функций (164). При рассмотрении отдельных тестов (пример ниже) или даже отдельных функций разные алгоритмы достигают разных максимумов производительности, а также приближаются к ним существенно по-разному.

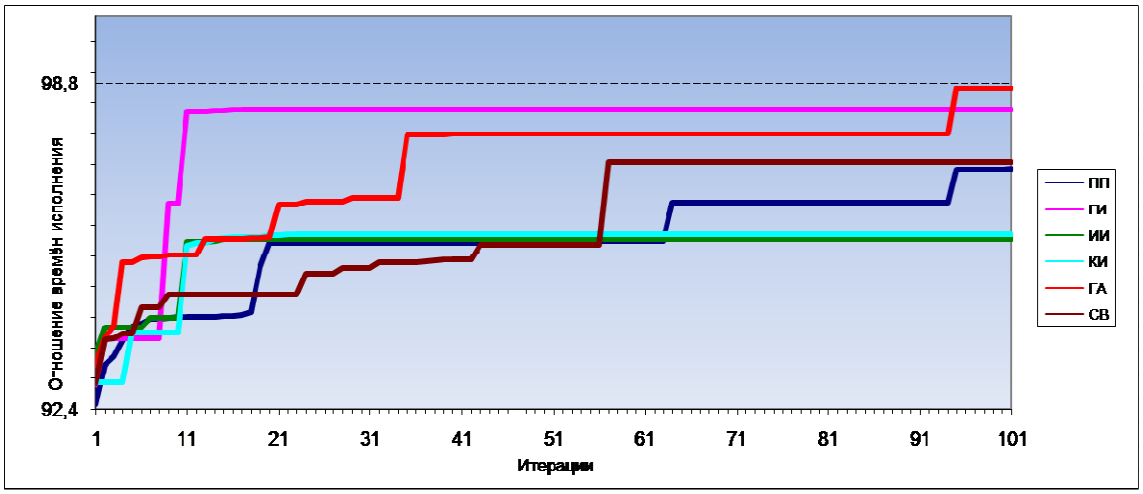

Рисунок 2. График зависимости от итерации суммарного времени исполнения всех функций теста CPU2000 SPEC 186.crafty для каждого из алгоритмов.

Если анализировать итеративный процесс, можно заметить, что алгоритмы статистического выбора и полного перебора с приоритетом приближаются к оптимальному результату позже всех. Для первого – это связано с фазой пробных запусков, в процессе которых идёт накопление информации, для второго – с фиксированным порядком перебора не являющимся наиболее оптимальным для широкого круга функций. Таким образом, эти алгоритмы буду давать тем более точный результат, чем больше запусков будет произведено. Алгоритмы итеративного и комбинированного исключения раньше всех достигли значений близких к лучшим найденным, что при сильно ограниченном количестве итераций позволяет им показывать наилучшие результаты. Алгоритм группового исключения показал наихудший результат и остановил работу уже на 16 итерации, однако в некоторых случаях он может становиться лидером.

Например, рассмотрим такой же график для приложения 186.crafty (Рисунок 2), которое представляет собой шахматную программу. На графике видно, что наилучший результат показал генетический алгоритм, однако значительно раньше вышел в лидеры и долго им оставался (86 итераций) алгоритм группового исключения, который на общем графике занимал последнюю позицию; в то же время алгоритм итеративного исключения уступил лидеру почти 2,95%. Полученный результат обусловлен очень слабым влиянием опций друг на друга в данном приложении, что в общем случае выражается не так явно.

Наибольшее относительное уменьшение времени в 11.3% было получено алгоритмом итеративного исключения для приложения 187.facerec – исходный код для программы распознавания лиц, который был наиболее восприимчив к выбранным опциям оптимизаций. Почти все тесты (25 из 26) показали прирост производительности более 1%. Распределение прироста производительности по тестам показано на графике (Рисунок 3).

6. Заключение

Проведенные эксперименты по влиянию шести опций управления оптимизациями компилятора на производительность показывают жизнеспособность данного подхода для широкого набора приложений. Для отдельных тестов прирост производительности в 8-11% достигается уже на 10 итерациях построения и запуска при использовании алгоритмов исключения. В среднем необходимо около 40 итераций для улучшения суммарной производительности на 3-4%.

В дальнейших работах будет в десятки раз расширен набор опций компилятора, а также опробовано применение самообучающейся экспертной системы, способной на основе свойств приложений существенно сократить число необходимых итераций вплоть до одной - статической компиляции.

7. Список литературы

[1] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimization space", *In Proc. ACM Workshop on Profile*

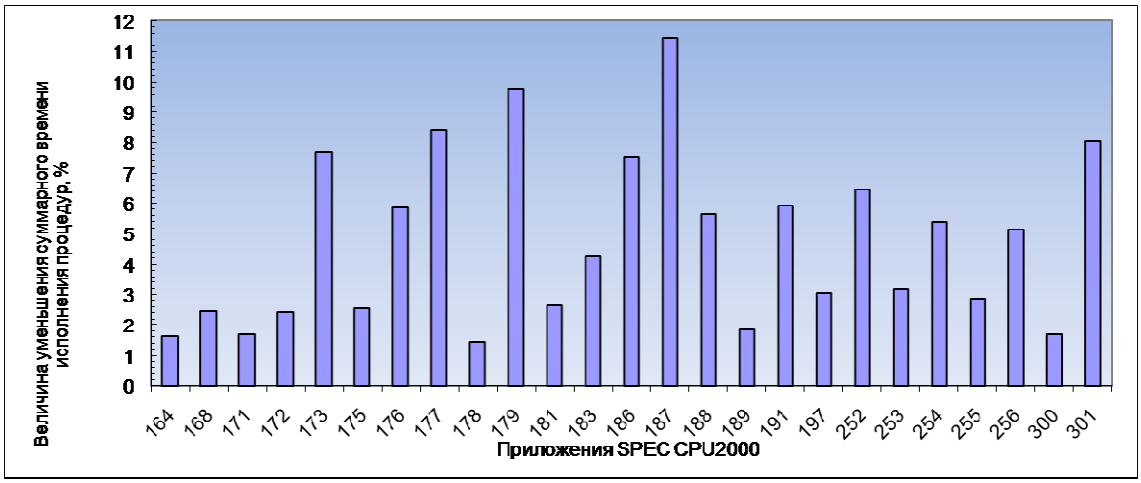

Рисунок 3. График прироста производительности для отдельных тестов SPEC CPU2000

and Feedback Directed Compilation, 1998, Organized in conjunction with PACT98.

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

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

[4] M. Stephenson and S. Amarasinghe, "Prediction unroll factor using supervised classification", *In ERRR/ACM International Symposium on Code Generation and Optimization (CGO 2005),* ERRR Computer Society, 2005.

[5] Yom-Toy, J. Thomson, O. Temam, A. Zaks, H. Leather, C. Miranda, M. Namolaru, E. Bonilla, Saclay, B. Mendelson, C. Williams, Haifa, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, F. Bodin "MILEPOST GCC: machine learning based research compiler", *ARC, International, UK, CAPS Enterprise, France*, 2007.

[6] K. Hoste, L. Eeckhout, "COLE: Compiler Optimization Level Exploration", *ELIS Department, Ghent University, Sing-Pietersnieuwstraat 41, B-9000 Gent, Belgium*, 2008.

[7] K. Deb, "Multi-Objective Optimization using Evolutionary Algorithms", *Wiley*, 2001.

[8] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam, "MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization", *ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France*, 2007.

[9] M. Byler, M. Wolfe, J.R.B. Davies, C. Huson, and B. Leasure, "Multiple version loops." *In ICPP*, 1987, pages 312-318, 2005.

[10] K. D. Cooper, M. W. Hall, and K. Kennedy, "Procedure cloning", *In Proceedings of the 1992 IEEE International Conference on Computer Language*, pages 99-105, 1992.

[11] P. Diniz and M. Rinard. "Dynamic feedback: An effective technique for adaptive computing", *In Proc. PLDI*, pages 71-84, 1997.

[12] G. Fursin, C. Miranda, S. Pop, A. Cohen, O. Temam, "Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation", *Alchemy group, INRIA Futurs and LRI, Paris-Sud 11 University, Orsay, France*, 2007.

[13] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *In ACM SIGPLAN Notices*, 2000.

[14] R. H. Saavedra and D. Park, "Improving the effectiveness of software prefetching with adaptive execution", *In Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.

[15] M. Voss and R. Eigemann, "High-level adaptive program optimization with adapt", *In Proceedings of the Symposium on Principles and practices of parallel programming*, 2001.

[16] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, "A Practical Method For Quickly Evaluating Program Optimizations", *Institute for Computing Systems Architecture, University of Edinburgh, UK*, 2005.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", *In 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] J. Lau, S. Schoenmackers, and B. Calder, "Transition phase classification and prediction", *In International Symposium on High Performance Computer Architecture*, 2005.

[19] Z. Pan, R. Eignmann, "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning.", *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[20] S. Triantafyllis, M.J. Bridges, E. Raman, G. Ottoni and D. August, "A Framework for Unrestricted Whole-Program Optimization.", *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[21] Z. Pan, R. Eignmann, "Fast, Automatic, Procedure-Level Performance Tuning.", *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[22] H. Feltl, "Ein Genetischer Algorithmus fuer das Generalized Assignment Problem", *Diplomarbeit*, 2003.

[23] M. Haneda, P. Knijnenburg, H. Wijshoff "Automatic Selection of Compiler Options Using Non-Parametristic Inferential Statistics.", *Proceedings of the 14th International Conference on Parallel Architecture and Compilation Techniques*, 2005.