

# Поддержка процесса повышения производительности компиляторов

Ю.В. Баскаков, В.Ю. Волконский, А.В. Грабежной, М.И. Нейман-заде, Л.Г. Тарасенко

**Аннотация.** Работа посвящена проблеме обеспечения качества оптимизирующих компиляторов. В частности, рассматривается деятельность по поддержке процесса разработки, направленная на совершенствование способности компилятора создавать высокопроизводительный целевой код. На основе результатов, полученных авторами в рамках проекта по созданию оптимизирующего компилятора, приводится оценка некоторых аспектов этой деятельности.

## Введение

Быстродействие целевого кода является одной из важнейших характеристик качества компилятора. Она определяет, насколько эффективно компилятор способен распознавать статические и динамические свойства исходной программы и проецировать их на возможности целевой архитектуры.

Компилятор может применять десятки оптимизирующих преобразований, которые должны давать предельно быстрый для данной программы целевой код [8]. Повышение производительности должно обеспечиваться за счет усиления оптимизирующих возможностей компилятора, которое состоит как в расширении набора оптимизирующих преобразований, поддерживаемых компилятором, так и в устранении дефектов в реализованных ранее компонентах.

Очевидно, что в обоих случаях требуется принять ряд мер аналитического характера по выявлению неоптимальных фрагментов в целевом коде (в дальнейшем обозначаемых как «неоптимальности»). Существенно и то, что, как показывает практика, приемлемого уровня быстродействия можно добиться лишь в результате непрерывного контроля производительности. Действительно, любые модификации исходного кода компилятора, будь то исправление ошибок или расширение функциональности, являются потенциальными источниками деградации достигнутых показателей быстродействия.

И, наконец, как при анализе кода для выявления неэффективности, так и при контроле производительности, требуется некое представительное множество программ. Его формирование является еще одной проблемой, которую необходимо решить для достижения желаемого результата.

Таким образом, для повышения производительности целевого кода помимо непосредственного внесения изменений в код компилятора чрезвычайно важна деятельность по поддержке этого процесса. Она состоит в выработке предложений по развитию компилятора и контроле этого развития. В статье рассматриваются вопросы, связанные с этой деятельностью, описывается опыт авторов по ее реализации в процессе разработки оптимизирующего компилятора для архитектуры «Эльбрус».

## 1. Выработка предложений по развитию компилятора

В данном разделе рассматриваются возможные причины потерь производительности при исполнении целевого кода, которые должны устраняться в ходе разработки компилятора. Будут изложены подходы к выявлению неоптимальностей целевого кода.

### 1.1. Причины неоптимальностей

Среди основных причин построения компилятором неоптимального кода можно выделить следующие:

- неполнота совокупности условий, определяющих применимость оптимизации;
- несовершенство аналитических компонент компилятора;
- ошибки в сборе аналитической информации;
- излишняя агрессивность или консерватизм оптимизации;
- отсутствие ряда статистически значимых оптимизирующих преобразований;
- неоптимальный выбор последовательности преобразований;
- неоптимальность алгоритмов планирования кода на параллельные аппаратные ресурсы.

Рассмотрим эти факторы с точки зрения их внешнего проявления.

#### Несрабатывание реализованной в компиляторе оптимизации в контексте применимости

Поддерживаемое компилятором оптимизирующее преобразование может не быть выполнено по следующим причинам:

- компилятор не распознает текущий контекст в качестве контекста исполнения оптимизации, т.е. имеет место неполнота системы условий применимости (например, преобразование *Dead Code Elimination* [6], которое не считает операции записи в локальный массив «мертвыми» операциями);
- в результате анализа принимается неправильное решение о свойствах объектов или отношениях, в которых состоят эти объекты (наличие или отсутствие зависимостей, принадлежность одному классу эквивалентности и так далее), то есть имеют место ошибки при сборе аналитической информации (например, преобразование *Memory Access Widening* [6] может не срабатывать из-за того, что некорректно работает анализ, распознающий выравнивание адресов обращений в память);
- анализ не может дать определенный ответ на поставленный вопрос, т.е. имеет место несовершенство алгоритмов или ограниченность возможностей анализа (например, индексный анализ может быть ограничен только на линейные выражения);
- оценка эффективности либо груба, либо не учитывает важных метрических характеристик; в этом случае стоит говорить об излишнем консерватизме эвристик применения оптимизации (например, преобразование *Loop Interchange* [2], изменяющее порядок следования циклов в плотном гнезде, в качестве критерия применимости может учитывать только случай, когда число итераций внутреннего цикла меньше числа итераций внешнего, и не применяться, если эти значения равны, не учитывая таким образом возможного выигрыша от увеличения локальности данных);
- происходит неправильный отказ от применения в условиях ограничений, влияющих на допустимое количество срабатываний оптимизации (например, если оптимизация является агрессивной по отношению к росту кода, то на этот показатель может быть наложено ограничение, так что из нескольких контекстов применимости приходится выбирать тот, где выгоднее применить преобразование).

#### Неэффективное применение оптимизации

В ряде случаев применение оптимизирующего преобразования в допустимом контексте приводит к нежелательным потерям производительности. По этой причине может потребоваться запрет преобразования. Подобные ситуации возникают вследствие проблем с оценкой эффективности преобразования (оптимизация слишком агрессивна), либо из-за ошибок при сборе или использовании аналитической информации.

В этом ряду присутствуют не только неэффективные применения оптимизации как таковые, но и применения с не лучшим образом выбранным параметром. Скажем, преобразование *Loop Unroll-*

*ing* [2], приводящее в некой ситуации к потерям, все же может иметь смысл, но при раскрутке цикла на иное количество итераций.

### **Преобразование не поддерживается**

Ряд полезных преобразований, которые в принципе дали бы эффект на статистически значимом наборе анализируемых задач, компилятором могут не поддерживаться. Например, возможно, что не используются те или иные свойства архитектуры или системы команд. При внесении предложений по реализации новых преобразований должна оцениваться их важность.

### **Испорченный контекст применимости преобразования**

Правильная последовательность применения преобразований тоже существенна для достижения максимального эффекта от оптимизаций. Так, определенные оптимизации могут расширять или даже создавать контекст применимости для последующих. Напротив, предшественник может разрушать благоприятный контекст (например, *Loop Unrolling* может разрушить контекст *Loop Fusion* [2]). Неправильно выстроенная линейка оптимизаций может приводить к серьезным потерям.

### **Неэффективно спланированный код**

Еще один важный аспект – это планирование и распределение ресурсов. Помимо неоптимальных алгоритмов на эффективность планирования могут влиять ложные зависимости (из-за ошибок анализа) или неправильный выбор критического ресурса. Стоит отметить, что ряд оптимизаций в результате своей работы могут влиять на критичность того или иного ресурса, так что причины неэффективности планирования следует искать и на более ранних фазах.

## **1.2. Подходы к выявлению неоптимальностей**

В проекте были использованы два дополняющих друг друга подхода: анализ производительности на представительном классе задач и исследование оптимизаций.

Анализ производительности основан на идее существования некоторого представительного класса задач, которые достаточно хорошо фиксируют характер и особенности исполняемых на целевой платформе приложений [11]. Таким образом, должно быть сформировано множество задач, на котором будет оцениваться работа компилятора.

Целью анализа производительности является получение ответов на следующие вопросы: какие оптимизирующие преобразования должны быть реализованы в компиляторе, в каких ситуациях эти преобразования следует применять и какова должна быть последовательность их применения. В результате анализа может быть обнаружена необходимость как в принципиально новых преобразованиях, так и в уточнении контекста применимости реализованных ранее. Важен и теоретический прогноз на представительном классе задач. Нужно знать, какие резервы архитектуры остаются неиспользованными, чтобы выделить наиболее актуальное направление развития. Знание предельного показателя является также важным психологическим фактором и стимулом к дальнейшему улучшению. Он позволяет судить, на какой стадии в текущий момент находится разработка и к чему нужно стремиться.

Чтобы определить границы поддерживаемой функциональности, целесообразно проводить исследование реализованных в компиляторе оптимизаций, в частности, для расширения области применимости. Исследование должно проводиться с использованием специализированных тестовых комплектов, реализующих достаточно полное покрытие условий применимости оптимизаций и позволяющих производить количественную оценку эффекта от их применения.

От анализа производительности на представительном классе задач, который также способен выявлять направления расширения условий применимости оптимизаций, исследование преобразований отличается своим систематическим характером. Действительно, целью исследований является оценка эффекта от оптимизации при переборе различных контекстов. В свою очередь, анализ производительности способен обнаруживать необходимость в принципиально новой функциональности.

Для краткости рассмотрим лишь процесс анализа производительности.

### 1.3. Анализ производительности

С целью анализа нами были использованы задачи из пакета SPEC [12], фрагменты мультимедийных библиотек и приложения по кодированию/декодированию видео-изображений, а также - подборка цикловых эталонов. В состав последних вошли программы, достаточно хорошо представляющие многообразие цикловых структур, но не содержащие сложных зависимостей. Это позволило сконцентрироваться на отладке базовых принципов работы с циклами. Мультимедийные приложения, в свою очередь, являются хорошей базой для анализа способности компилятора использовать потенциал операций по работе с упакованными целыми и вещественными числами. Наконец, пакет SPEC давно зарекомендовал себя как набор задач, отражающих основные тенденции программирования и позволяющих проверить возможности архитектуры.

#### 1.3.1. Структура процесса анализа производительности

Анализ производительности (Рис.1) представляет собой процесс, который получает на вход набор исходных задач, а на выходе выдает набор предложений по повышению производительности компилятора. Этот набор доступен участникам процесса разработки компилятора через архив предложений. Предложения по повышению производительности попадают в архив предложений либо напрямую, либо через временный архив предложений. Последний выполняет роль буфера, куда помещаются предложения, значимость которых требует подтверждения в ходе дальнейшего анализа, а также значимые трудоемкие предложения, приоритетность которых еще не установлена. Между участниками процесса анализа и участниками процесса разработки имеется двусторонний канал общения, служащий для координации проводимых работ. Процесс анализа взаимодействует с системой контроля деградаций, поставляя ей эталонные оценки по задачам. Система, в свою очередь, выдает регулярные отчеты о состоянии производительности задач, стоящих на контроле. Наконец, еще одной функцией анализа производительности является формирование представительного множества значимых фрагментов для постановки на контроль.



Рис. 1. Структура процесса анализа производительности

#### 1.3.2. Стратегии анализа производительности

Анализ производительности на некотором наборе приложений существенно зависит от специфики этого набора. Так, в случае однотипных тестовых примеров анализ состоит в изучении границ области применимости и исследования качества реализации в компиляторе некоторой фикси-

рованной цепочки преобразований. В то же время, целью анализа производительности на небольшом наборе реальных приложений (отражающих требования потенциального пользователя) является как исследование качества компилятора в целом, так и выявление новых цепочек оптимизаций, позволяющих создавать наиболее эффективный код для отдельных значимых фрагментов каждого приложения.

1.3.2.1. Классификация предложений и приоритеты поиска неоптимальностей

Рассмотрим различные типы предложений по улучшению оптимизирующих возможностей компилятора:

1. предложения языкового уровня (предложения по учету соглашений или специфичных свойств, сформулированных в стандарте языка программирования; например, было обнаружено, что для задач, написанных на языке Fortran 77 [1], компилятор не учитывал требование, чтобы индексные выражения массива находились в рамках своих верхних и нижних границ по соответствующим измерениям, а также - ограничения на ассоциации, предполагая тем самым наличие лишних зависимостей);

2. предложения по межпроцедурным оптимизациям (например, *Procedure Inlining* [2], *Procedure Cloning* [2]);

3. предложения по нецикловым оптимизациям, меняющим структуру управления (например, *Conditional Branches Optimization* [7]);

4. предложения по цикловым макрооптимизациям (например, *Loop Interchange*, *Loop Fusion* или *Loop Unrolling*);

5. предложения по потоковым и цикловым оптимизациям, не меняющим структуру управления (например, *Dead Code Elimination*, *Redundant Load Elimination* [6] или *Common Subexpression Elimination* [6]);

6. предложения по оптимизациям, специфичным для архитектуры (в случае E2K предложения данного типа затрагивали такие возможности архитектуры, как аппаратная поддержка конвейеризованных циклов, полная аппаратная поддержка предикатных вычислений, механизм разрешения конфликтов по обращению в память и др. [4]).

Этот порядок определяет систему приоритетов при поиске неоптимальностей и формировании предложений. Он согласуется с последовательностью оптимизаций, применяемых компилятором к исходному коду программы.

Таким образом, при работе с задачей желательно сначала сформулировать все значимые предложения языкового уровня, затем – предложения по межпроцедурным оптимизациям и так далее в порядке возрастания нумерации списка.

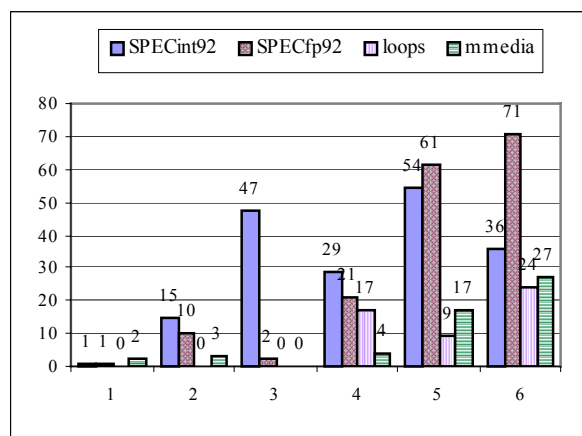


Рис.2. Распределение количества предложений по типам для разных групп задач

В заключение приведем статистику по предложениям, собранную за год с момента начала анализа производительности компилятора. На Рис.2 показано распределение количества выработанных предложений по типам согласно приведенной выше классификации. Рассматриваются четыре группы задач – SPECint92, SPECfp92, цикловые эталоны (loops) и мультимедийные приложения (mmedia).

Анализ производительности на этом классе задач позволил выявить существенные и статистически значимые преобразования, которые еще не были реализованы в компиляторе, а также помог подобрать правильные эвристики эффективности в

контексте реальных приложений и расширить контекст применимости уже существующих оптимизаций.

Ниже мы рассмотрим процесс анализа производительности и формирования предложений по оптимизациям в том виде, в котором он применялся на практике.

### 1.3.2.2. Анализ производительности на отдельной задаче

Особенность большинства задач, используемых нами для анализа, состоит в том, что основное время их работы приходится на сравнительно небольшую часть кода (как правило, ~10%). В подобных случаях достижение максимума производительности невозможно без оптимальной работы компилятора именно на этих частях. Выявление таких «горячих» фрагментов для всей задачи, исследование достигнутой на них производительности и возможности ее увеличения, составляют специфику анализа производительности на используемых нами приложениях. Опыт реализации упомянутых выше идей на практике позволил сформулировать общую концепцию анализа производительности на классе задач, которая и будет представлена в данном разделе.

Рассмотрим основные этапы анализа производительности на отдельно взятой задаче.

**Выбор фрагментов для анализа.** Достаточно ограничить список рассматриваемых процедур теми, которые составляют верхушку профиля исполнения задачи. Анализ будет наиболее эффективным, если на выбранные процедуры приходится подавляющая часть (80-90%) времени работы задачи. В каждой из процедур необходимо выделить набор фрагментов, определяющих основное время работы процедуры. Здесь основным критерием является прозрачность для анализа (небольшой размер фрагмента) и его замкнутость относительно управления (одна точка входа).

**Исследование фрагментов.** Для каждого из фрагментов желательно найти такое его преобразование, которое обеспечивало бы максимум производительности. При поиске должны быть учтены как специфика фрагмента, так и все ограничения и доступные ресурсы целевой архитектуры.

Для каждого фрагмента  $\Phi$ , в зависимости от его типа, устанавливается порядок действий по поиску оптимального преобразования:

#### 1. $\Phi$ - многоитерационный самый вложенный цикл с простым управлением;

(а) Если отсутствует межитерационная рекуррентность (или она не критична), то оптимизация должна быть направлена на уменьшение числа операций в цикле и максимальное использование возможностей архитектуры, таких как, например, потенциал широкой команды, аппаратная поддержка конвейеризации циклов, комбинированные операции, упреждающее чтение элементов массива, операции над упакованными целыми и вещественными числами и т. д. Здесь могут оказаться полезными *Loop Unrolling*, *Memory Access Widening*, а также ряд платформозависимых оптимизаций. При этом должны учитываться ресурсные ограничения (например, размер и свойства кэш-памяти, количество доступных регистров, размер широкой команды). Так что к упомянутым выше оптимизациям могут добавиться еще и методы для борьбы с ограничениями: *Prefetch* – для оптимизации работы с кэш-памятью, *Loop Distribution* [2] – при нехватке регистров и т. д.

(б) Если наличие рекуррентности в цикле мешает эффективному использованию всех предоставляемых архитектурой возможностей, то поиск оптимального преобразования должен быть направлен на уменьшение длины рекуррентности (*Lazy Code Motion* [5], *Expression Balancing* [6] и др.) или на ослабление ее роли. Последнее подразумевает возможность использования свободных устройств операциями, пришедшими извне, что требует анализа всего содержащего цикл региона или гнезда циклов (3);

#### 2. $\Phi$ - многоитерационный цикл со сложным управлением (возможно, содержит вложенные малоитерационные циклы):

Необходимо обратить внимание на использование всех резервов упрощения управления (*Loop Nesting* [2], *Loop Unswitching* [2], *Loop Splitting* [2], *Procedure Inlining* и т.д.) Если это удалось, то дальше действовать согласно (1), иначе – выделить наиболее важные ациклические области и разбираться с ними по отдельности (4);

3.  $\Phi$  - гнездо циклов с простым управлением (или регион, содержащий несколько однотипных циклов):

Необходимо проанализировать применимость гнездовых оптимизаций (*Loop Fusion, Unroll-and-Jam* [3], *Loop Interchange* и т. д.) с целью привести по возможности все внутренние циклы к виду (1а);

4.  $\Phi$  - ациклический участок, содержащий малоитерационные циклы:

Если таким участком является вся процедура, то полезно исследовать возможность inline-подстановки (процедура небольшая) или частичной inline-подстановки (можно отделить наиболее вероятную часть потока управления). В общем случае, объектом исследования является набор небольшого числа наиболее вероятных траекторий потока управления такой, что суммарная вероятность пучка выбранных траекторий вносит основной вклад в весь поток управления (>80%) и ставится задача исследовать возможность минимизации критических путей, отвечающих этим траекториям.

После того, как найден способ оптимизации фрагмента, необходимо произвести его разложение в цепочку оптимизаций, которые уже реализованы (или могут быть реализованы) в компиляторе. Во время этого процесса возможна значительная корректировка найденного преобразования. Возможная цепочка оптимизаций может не быть единственной.

**Выработка предложений.** Если код, полученный компилятором, проигрывает в сравнении с результатом теоретического исследования, то путем последовательных проверок на рассматриваемом фрагменте оптимизаций из построенной цепочки находится неэффективно отработавшая (как вариант – отсутствующая) оптимизация и оформляется предложение с указанием на эту неэффективность и методы ее устранения.

### 1.3.3. Примеры предложений

**Пример 1.** *Задача 026.compress - уменьшение рекуррентности в самом вложенном цикле процедуры compress.*

<p>а) Исходный цикл:</p> <pre> probe:   if ( (i -= disp) &lt; 0 )     i += hsize_reg;    if ( htabof(i) == fcode ) {     ent = codetabof(i);     continue;   }    if ( (long)htabof(i) &gt; 0 )     goto probe;                     </pre>	<p>б) Преобразованный цикл:</p> <pre> i1 = i - disp; probe:   i = i1;   if ( (i1) &lt; 0 )     i1 += ( hsize_reg - disp);   else     i1 -= disp;    if ( htabof(i) == fcode ) {     ent = codetabof(i);     continue;   }    if ( (long)htabof(i) &gt; 0 )     goto probe;                     </pre>
--	---

Рис.3. Уменьшение рекуррентности для цикла из процедуры compress

Среднее число итераций рассматриваемого цикла (Рис.3 а) достаточно для эффективного использования аппаратной конвейеризации, но присутствующая в цикле рекуррентность:

```

if ( (i -= disp) < 0 )
  i += hsize_reg;
                    
```

не позволяет полностью использовать ресурсы, предоставляемые широкой командой.

Результатом анализа производительности для данного цикла стало предложение об уменьшении рекуррентности путем ее балансировки:

```
if ( (i1) < 0 )
    i1 += (hsize_reg - disp);
else
    i1 -= disp;
```

Применение балансировки стало возможным только после переноса одной из операций, входящих в рекуррентную цепочку, по обратной дуге цикла (*Lazy Code Motion*). В результате рассмотренного преобразования длина рекуррентности в данном цикле уменьшилась с 4-х до 3-х тактов. Таким образом, на данном примере была обнаружена эффективность цепочки преобразований:

*Lazy Code Motion* → *Expression Balancing* → *Аппаратная Конвейеризация Циклов*  
для самых вложенных циклов с большим числом итераций, содержащих межитерационную рекуррентность.

**Пример 2.** *Задача 023.eqntott - векторизация вычислений для основного цикла задачи.*

<p>а) Исходный цикл:</p> <pre>for (j = 0; j &lt; ninputs; j++) {     aa = i[0]-&gt;ptand[j];     bb = bb_[j];     if (aa == 2)         aa = 0;     if (aa != bb) break; }</pre>	<p>б) Цикл после <i>MAW</i>:</p> <pre>for (j = 0; j &lt; (ninputs/4); j++) {     aa_64 = ((long long*)(i[0]-&gt;ptand))[j];     bb_64 = ((long long*)(bb_))[j];     if ((aa_64 &amp; 0xffff) == 2)         aa_64 &amp;= 0xfffffffffff0000;     if ((aa_64 &amp; 0xffff0000) == 0x20000)         aa_64 &amp;= 0xffffffff0000ffff;     if ((aa_64 &amp; 0xffff00000000) == 0x200000000)         aa_64 &amp;= 0xffff0000fffffff;     if ((aa_64 &amp; 0xffff000000000000) == 0x2000000000000)         aa_64 &amp;= 0x0000ffffffffff;     if (aa_64 != bb_64) break; }</pre>
<p>в) Цикл после всех преобразований:</p> <pre>for (j = 0; j &lt; (ninputs/4); j++) {     aa_64 = ((long long*) (i[0]-&gt;ptand))[j];     bb_64 = ((long long*)(bb_))[j];     aa_64 &amp;= pcmpeqh(aa_64, 0x2000200020002);     if (aa_64 != bb_64) break; }</pre>	

Рис.4. Векторизация вычислений

16-битный размер используемых в цикле (Рис. 4а) операций обращения к памяти позволил применить для данного цикла оптимизацию *Memory Access Widening (MAW)* с последующим объединением 4-х выходов из цикла по неравенству:

```
if (aa != bb) break;
```

в один (Рис. 4 б).

Векторный характер операций сравнения и присваивания позволил перейти к операциям над упакованными целыми (*MMX*) и наиболее эффективно использовать в дальнейшем аппаратную конвейеризацию для полученного цикла (Рис.4 в).

Результатом анализа производительности на данном примере было предложение о расширении применимости оптимизации *MAW* на циклы с несколькими выходами с последующим их объединением. Такое преобразование сделало возможным использование операций над упакованными целыми, что позволило применить аппаратную конвейеризацию наиболее эффективно (после преобразования время работы рассмотренного цикла уменьшилось на 45%). При этом повысилась согласованность работы связки оптимизаций *MAW* и *Использование MMX* в рамках последовательности оптимизаций:

*Memory Access Widening* → *Использование MMX* → *Аппаратная Конвейеризация Циклов*, для циклов с простым управлением и большим числом итераций.

**Пример 3.** *Задача 022.li - расширение inline-подстановки на класс процедур с переменным числом параметров.*

В результате анализа процедур задачи 022.li составляющих 90% профиля исполнения задачи:

%time	%summ	time	calls	time/call	name
24.96%	24.96%	609382743	11355441	53	xleval
23.13%	48.08%	564715946	2155566	261	mark__PARTINLINE_TAIL__
17.69%	65.77%	431900353	4516249	95	evform
15.52%	81.28%	378917528	12551992	30	xlsave
2.77%	84.05%	67571254	7403	9127	sweep
2.45%	86.50%	59754983	392421	152	xcond
2.19%	88.69%	53436181	665234	80	evfun
2.15%	90.84%	52613879	706186	74	binary
...					
Total time = 2441922835					

было обнаружено, что на небольшую процедуру *xlsave* с переменным числом параметром вызова, представляющую после оптимизаций один линейный участок, приходится значительная часть времени исполнения задачи. *Inline-подстановка* этой процедуры в места ее вызова не происходила из-за ограниченности применения оптимизации *Procedure Inlining* на процедуры с фиксированным числом параметров. Кроме того, заметная часть времени исполнения уходила на запись параметров в массив в месте вызова и последующее чтение в самой процедуре *xlsave*. Было предложено расширить область применения *Procedure Inlining* на процедуры с переменным числом параметров. После реализации этого предложения и применения *inline-подстановки* процедуры *xlsave* на задаче 022.li произошло растворение заметной части (более 50% процедуры *xlsave*) подставленного кода в местах вызова.

Верхушка профиля исполнения задачи 022.li после применения *Procedure Inlining* к процедуре *xlsave*:

%time	%summ	time	calls	time/call	name
27.13%	27.13%	609382743	11355441	53	xleval
25.14%	52.28%	564715946	2155566	261	mark__PARTINLINE_TAIL__
24.76%	77.05%	556087470	4516249	123	evform
4.06%	79.30%	91254668	392421	232	xcond
3.56%	83.05%	80139720	665234	120	evfun
3.00%	84.05%	67571254	7403	9127	sweep
2.34%	90.84%	52613879	706186	74	binary
...					
Total time = 2245435678					

В данном примере эффект от предложенной оптимизации составил 8% от времени исполнения всей задачи.

**Пример 4.** *Цикл с потенциальным конфликтом «запись-чтение» - использование аппаратного механизма динамического разрешения конфликтов по обращению в память (DAM) и перенос глобалов на регистры.*

В рассматриваемом цикле (Рис. 5 а) переносу глобальных объектов *glob\_p* и *glob\_w* на регистры мешает потенциальный конфликт с операцией записи по нелокализуемому адресу

(*\*glob\_p = ...*). Было предложено разорвать потенциальные зависимости вида «запись-чтение» с помощью аппаратно-поддерживаемого механизма DAM и после этого осуществить перенос.

<p>а) Исходный цикл:</p> <pre>do {   if ( --glob_w &lt; 0 ) break;   *glob_p = *(--stackp);   (int*) glob_p++; } while ( stackp &gt; de_stack );</pre>	<p>б) Преобразованный цикл:</p> <pre>loc_w = lock (glob_w); loc_p = lock (glob_p);  do {   if ( --loc_w &lt; 0 ) break;   *loc_p = *(--stackp);   check (glob_w);   check (glob_p);   if ( %MLOCK ) goto recovery_code; cont:   (int*) loc_p++; } while ( stackp &gt; de_stack );  glob_w = loc_w; glob_p = loc_p; ... recovery_code: /* запись в память правильных значений глобальных объектов */ glob_w = loc_w; glob_p = loc_p; *loc_p = *stackp; /* повторная запись по адресу loc_p */ loc_w = lock (glob_w); loc_p = lock (glob_p); goto cont;</pre>
--	---

Рис.5. Использование аппаратного механизма разрешения конфликтов по обращению в память и перенос глобалов на регистры

На Рис. 5 *lock(obj)* обозначает операцию чтения по адресу *&obj* с одновременной установкой контроля за модификациями области памяти по этому адресу; *check(obj)* обозначает проверку отсутствия модификации области памяти по адресу *&obj* с момента постановки на контроль. В случае установления факта модификации, операцией *check(obj)* выставляется флаг %MLOCK, по которому производится переход на компенсирующий код.

В результате преобразования межитерационная рекуррентность составила 2 такта, тогда как в исходном цикле рекуррентность составляла 5 тактов.

## 2. Контроль вносимых изменений

Внесение изменений в исходный код компилятора является потенциальным источником ошибок и, следовательно, может негативно сказываться на достигнутых показателях производительности. Поэтому при любых попытках модифицировать код компилятора целесообразно оценивать эффект планируемого изменения на производительность и, в зависимости от результатов оценки, разрешать это изменение или запрещать.

Оценка эффекта модификаций на производительность должна обладать свойством *достоверности*. В идеале она должна основываться на результатах реальных замеров быстродействия, проводимых на всем множестве программ, которые может воспринять на вход компилятор. Однако в силу потенциальной бесконечности такого множества область оценки должна быть ограничена на достаточно представительный класс задач. Стоит отметить, что даже в этих условиях для получения *полной* картины о состоянии производительности могут потребоваться часы или даже сутки (особенно в условиях, когда вместо реального процессора используются программные модели, существенно замедляющие скорость выполнения, а пользовательские характеристики компилятора, такие как время компиляции, еще далеки от промышленных требований), т. к. в представительный класс обычно попадают достаточно большие и ресурсоемкие задачи. Так что в процессе разработки, когда действуют временные ограничения и требуется *оперативность* принятия решений, подобные методы оценки могут оказаться практически неосуществимыми.

В рамках данного проекта функция контроля вносимых изменений была возложена на систему контроля деградаций. При этом в качестве набора задач, на которых осуществлялся контроль, использовались задачи, участвовавшие в процессе анализа производительности. Чтобы обеспечить оперативность оценки эффекта модификаций на производительность и, в то же время, сохранить свойство полноты, процесс контроля деградаций был разделен на *оперативную* и *регулярную* составляющие.

Оперативный контроль деградаций использует быстрые, но не всегда адекватные, метрики [9] для оценки эффекта предполагаемых изменений. Регулярный же контроль призван частично компенсировать возможные нежелательные последствия ослабленного оперативного контроля. Он основывается на результатах замеров производительности стоящих на контроле задач и проводится с заданной периодичностью. Как оперативный, так и регулярный контроль используют принципы регрессионного тестирования, производя сравнения новых показателей быстродействия с полученными ранее. По результатам сравнений с учетом утвержденных критериев может быть принято решение о запрете планируемых или даже об отмене произведенных ранее изменений.

Рассмотрим вкратце некоторые аспекты реализации оперативного контроля, уделив при этом особое внимание метрикам.

Для обеспечения свойства оперативности контроля деградаций были использованы две быстрые метрики оценки производительности: метрика оценки времени выполнения больших задач по времени выполнения наиболее значимых фрагментов и метрика срабатываний оптимизаций.

Для поддержки первой из упомянутых метрик из ряда больших задач, стоящих на контроле, были выделены и оформлены в виде отдельных тестовых примеров «горячие» фрагменты. При этом обязательными условиями были адекватность поведения фрагмента в контексте исходной задачи, а также уменьшение на несколько порядков времени его выполнения [10]. В качестве фрагментов, как правило, брались тела наиболее значимых функций с произведенными в нужных местах inline-подстановками. Исключение составила задача 047.tomcatv из пакета SPECfp92, где удалось выделить отдельные гнезда циклов, сохранив контекст. В таблице приведены данные о количестве выделенных фрагментов для задач из пакета SPEC92. Также для каждой задачи показано процентное отношение времени выполнения кода, соответствующего выделенным фрагментам, ко времени выполнения всей задачи без учета временных затрат на вызовы библиотечных функций. Стоит отметить, что адекватность поведения определяется не только статистикой посещений линейных участков рассматриваемого фрагмента, но и его окружением, которое при анализе должно представлять одинаковую сложность как в контексте тестового примера, так и в контексте задачи. К сожалению, добиться этого удавалось не всегда либо по причине чрезмерной трудоемкости реализации, либо в силу ограничения на время исполнения. Так что для некоторых задач наблюдается снижение достоверности оценки при трансляции с использованием анализа межпроцедурного уровня.

Помимо выделенных из больших задач тестовых примеров замер быстродействия при оперативном контроле осуществлялся и для избранных функций, входящих в состав мультимедийных библиотек, а также для цикловых эталонов.

В основе метрики срабатываний лежит предположение о том, что предельный уровень производительности на задаче не может быть достигнут без обязательного применения к ее коду некоторого набора преобразований. Он выявляется в процессе анализа и состоит из оптимизаций, значимых с точки зрения воздействия на производительность. Так что контроль деградаций может осуществляться по фактам *срабатываний* соответствующих преобразований, установленным на основе результатов компиляции задачи, и не требовать ее выполнения. Концептуально в исходном коде задачи выделяются некие контрольные точки, в каждой из которых при компиляции должен наблюдаться свой заранее установленный ряд оптимизирующих преобразований. О таких преобразованиях мы говорим как о *поставленных на контроль*<sup>1</sup>.

В третьем столбце таблицы приведены данные о количестве преобразований, признанных значимыми для задач из пакета SPEC92. При этом в скобках указано, сколько из них было поставлено на контроль (преобразования ставятся на контроль по мере появления у компилятора возможности для их отработки). В число значимых вошли преобразования, повышающие производительность более чем на 1%, а также преобразования, без которых этот эффект оказывается не таким весомым. Например, конвейеризация некоторого значимого цикла сама по себе может вносить заметный вклад в производительность задачи. Однако некая совокупность оптимизаций тела этого цикла может сделать эффект от конвейеризации еще более ощутимым. При этом эффект от отдельно взятого преобразования, входящего в совокупность, может быть и неразличим.

Был реализован сбор статистики срабатываний оптимизирующих преобразований. Она оформляется в виде *событий срабатывания*, регистрируемых в процессе компиляции программы. Каждое такое событие позволяет устанавливать срабатывание по идентификатору преобразования, а также именам исходного модуля и функции, номерам узлов, операций и строчек исходного кода программы.

Для контроля деградаций сформированы два репозитория: *репозиторий событий срабатывания оптимизаций* и *репозиторий временных показателей*.

В первом репозитории в качестве элементов хранятся шаблоны срабатываний, индексированные именами модулей программ. Они представляют собой тройку из идентификатора оптимизации, диапазона строчек исходного текста модуля, в котором должны происходить срабатывания, и количества срабатываний. При этом каждый из шаблонов может присутствовать в репозитории в одном из двух статусов:

- все срабатывания, описываемые шаблоном, имели место хотя бы однажды;
- срабатывания в полном объеме пока не происходили. Шаблоны срабатываний включаются в репозиторий на основе данных анализа производительности.

<sup>1</sup> Фактически, речь идет о постановке на контроль оптимизирующих преобразований вместе с контекстами их применения

	фрагменты		преобразования
<b>espresso</b>	10	(82%)	48 (42)
<b>li</b>	5	(78%)	73 (58)
<b>eqntott</b>	4	(85%)	14 (11)
<b>compress</b>	5	(98%)	27 (25)
<b>sc</b>	4	(62%)	68 (51)
<b>gcc</b>	11	(26%)	37 (31)
<b>wave5</b>	4	(76%)	93 (74)
<b>tomcatv</b>	5	(99%)	66 (64)
<b>alvinn</b>	3	(92%)	22 (21)
<b>ear</b>	5	(98%)	34 (24)
<b>swm256</b>	4	(99%)	54 (51)
<b>su2cor</b>	7	(99%)	125 (111)
<b>hydro2d</b>	3	(75%)	75 (66)
<b>nasa7</b>	5	(46%)	81 (63)
<b>fpppp</b>	2	(71%)	91 (59)

Во второй репозиторий заносятся текущие значения временных показателей программ. При этом, сохраняется информация как о текущем значении показателя, так и о наилучшем достигнутом ранее.

### 3. Результаты

Ниже приводятся результаты повышения производительности компилятора, которые достигнуты за год, прошедший с момента инициации процесса. К началу представленного периода в компиляторе были реализованы основные локальные и цикловые оптимизации, включая аппаратную поддержку конвейеризации циклов. В течение периода исследований функциональность компилятора была расширена за счет межпроцедурного анализа и поддержки механизма контроля обращений в память.

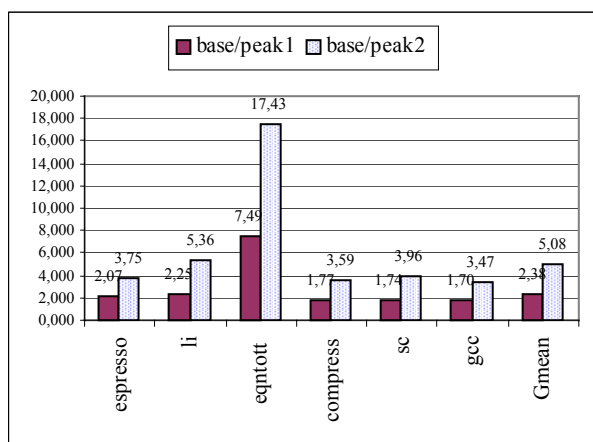


Рис.6. Эффект оптимизаций на задачах из пакета SPECint92 на начало (peak1) и конец (peak2) рассматриваемого периода

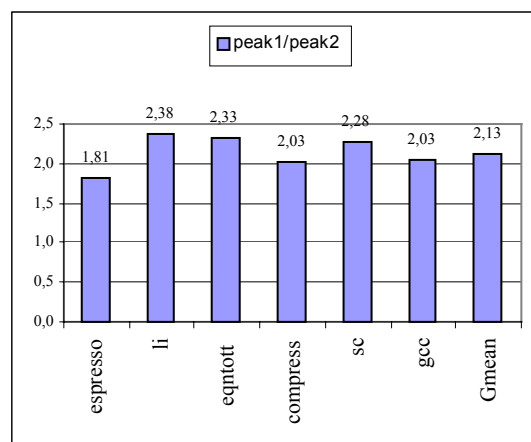


Рис.7. Ускорение на задачах из пакета SPECint92 за рассматриваемый период

На Рис.6 представлены относительные значения эффекта от оптимизаций (отношение показателей производительности на задачах, оттранслированных без оптимизаций – base, к пиковым показателям производительности – peak) на задачах из пакета SPECint92. Значение 17.43 на задаче 023.eqntott объясняется эффективным планированием с использованием аппаратной поддержки конвейеризации имеющегося в задаче многоитерационного цикла.

Рис.7 показывает достигнутое ускорение по каждой задаче по истечению периода исследований. Видно, что среднее геометрическое ускорение по всем задачам из пакета SPECint92 составило 2.13 при минимуме в 1.81 для задачи 008.espresso и максимуму в 2.38 на задаче 022.li.

Согласно Рис.9 среднее геометрическое ускорение по задачам из пакета SPECfp92 составило 4.88 при минимуме в 1.81 для задачи 094.frrrrr и максимуму в 7.70 на задаче 078.swm256.

### Заключение

В работе затронут ряд проблем, связанных с организацией процесса разработки оптимизирующего компилятора, ориентированного на построение высокопроизводительного целевого кода.

Рассмотрена деятельность, направленная на выработку предложений по развитию компилятора и обеспечение контроля вносимых в компилятор изменений. В частности, рассмотрен процесс анализа производительности компилятора на представительном классе задач, начиная с определения целей этого процесса и заканчивая рассмотрением возможной структурной схемы его взаимодействия с окружением. Для контроля вносимых в компилятор изменений было предложено распределение этой деятельности между оперативной и регулярной составляющими. В работе

пределение этой деятельности между оперативной и регулярной составляющими. В работе также приведены: классификация возможных причин неоптимальностей кода, создаваемого компилятором, стратегии анализа производительности, иерархия типов предложений по улучшению компилятора, определяющая стратегии поиска неоптимальностей, и примеры самих предложений.

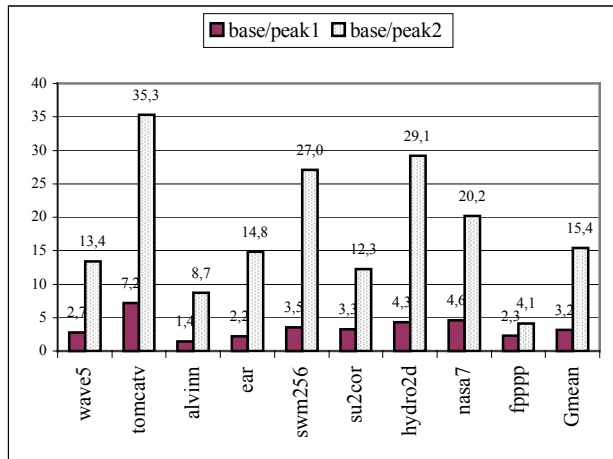


Рис.8. Эффект оптимизаций на задачах из пакета SPECfp92 на начало (peak1) и конец (peak2) рассматриваемого периода

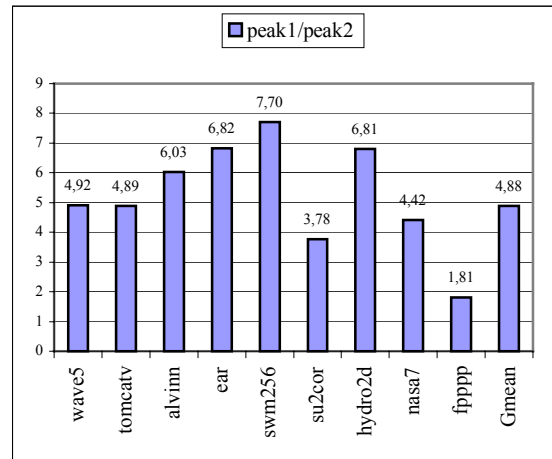


Рис.9. Ускорение на задачах из пакета SPECfp92 за рассматриваемый период

В результате практической реализации изложенных в работе принципов и идей были достигнуты средние ускорения в 2.13 и 4.88 раз на задачах из пакетов SPECint92 и SPECfp92 соответственно. При этом средний совокупный эффект от оптимизаций составил 5.08 для SPECint92 и 15.4 для SPECfp92. Стоит также отметить, что в ходе анализа производительности был сделан ряд предложений, позволивших уточнить параметры некоторых аппаратных механизмов.

Из возможных направлений дальнейшей работы стоит выделить улучшение методов оценки производительности компилятора с целью обеспечения оперативности и достоверности результатов независимо от уровня оптимизаций. Одним из решений может быть использование сохраненной аналитической информации для тестовых примеров, участвующих в оперативном контроле. Кроме того, предметом исследований в этом направлении может являться использование альтернативных подходов, таких как, например, статическая оценка производительности.

Отдельной проблемой, требующей внимательного изучения, является выбор представительного класса задач для проведения анализа производительности, а также множества задач для постановки на контроль. Особенно интересно было бы исследовать как в ходе развития компилятора меняется его производительность на задачах, не участвующих напрямую в процессе анализа.

## Литература

1. ANSI X3.9-1978 FORTRAN 77
2. Bacon D., Graham S., Sharp O., Compiler Transformations for High-Performance Computing, ACM Computing Surveys, 26(4), 1994, pp. 345-420
3. Carr S., Kennedy K., Improving the ratio of memory operations to floating-point operations in loops, ACM Transactions on Programming Languages and Systems, 16(6) 1994, pp.1768-1810
4. Diefendorf K., The Russians Are Coming: Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee. Microprocessor report, vol 2, num 2, 1999, pp. 7-11
5. Knoop J., Ruthing O., Steffen B., Lazy Code Motion, ACM SIGPLAN Notices, 27(7), 1992, pp.224-234
6. Muchnick S., Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997

7. Mueller F., Whalley D., Avoiding Conditional Branches by Code Replication, Proceedings of the 1995 ACM SIGPLAN conference on Programming language design and implementation, 30(6), pp 56-66
8. Ахо А., Сети Р., Ульман Дж. Компиляторы – принципы, технологии, инструменты, Вильямс, 2001
9. Баскаков Ю.В., Грабежной А.В., Лаврешников А.А., Рогов Р.Ю., Тарасенко Л.Г., Чернова Е.Ю., Вопросы организации системы обеспечения качества оптимизирующих компиляторов, Высокопроизводительные вычислительные системы и микропроцессоры, Сборник научных трудов ИМВС РАН, 2004
10. Лаврешников А.А., Пакет для оперативной оценки производительности оптимизирующих компиляторов, Высокопроизводительные вычислительные системы и микропроцессоры, Сборник научных трудов ИМВС РАН, 2002
11. Эйсымонт Л.К., Оценочное тестирование высокопроизводительных систем: цели, методы, результаты и выводы. Центр независимого межведомственного тестирования суперкомпьютерных систем. // Научная конференция «Параллельные вычисления и их применения», Черногловка, 2000
12. Standard Performance Evaluation Corporation - <http://www.spec.org>

**Баскаков Юрий Валерьевич.** Родился в 1976 году. Окончил Московский государственный университет им. М.В. Ломоносова в 1998 году. Автор 9 публикаций. Область научных интересов – компиляторы, анализ производительности, проектирование тестовых комплектов, FDT's. Ведущий научный сотрудник ЗАО МЦСТ.

**Грабежной Андрей Владимирович.** Родился в 1978 году. Окончил Московский государственный университет им. М.В. Ломоносова в 2002 году. Область научных интересов – оптимизирующие компиляторы, топология интегрируемых гамильтоновых систем. Старший научный сотрудник ЗАО МЦСТ.

**Нейман-заде Мурад Искендер оглы.** Родился в 1976 году. Окончил Московский государственный университет им. М.В. Ломоносова в 1998 году. Кандидат физико-математических наук с 2002 года, автор 10 научных работ. Область научных интересов – операторные модели в математической физике. Старший научный сотрудник ИМВС РАН.

**Тарасенко Лариса Григорьевна.** Родилась в 1947 году. Окончила Новосибирский государственный университет в 1970 году. Кандидат физико-математических наук с 1988 года, автор более 50 публикаций. Область научных интересов – тестирование и верификация компиляторов. Зав. отделом ЗАО МЦСТ.