DOI: 10.15514/ISPRAS-2025-37(4)-3

Определение неточностей в работе некоторых специализированных цикловых оптимизаций в компиляторе LCC для архитектуры «Эльбрус»

[1] *А.В. Ермолицкий*, ORCID: *0009-0008-2407-7420 <era@mcst.ru>*
[1,2] *Д.Н. Левченко,* ORCID: *0009-0004-6744-9889 <levchenko_d@mcst.ru>*
[1,2] *М.И. Нейман-заде,* ORCID: *0000-0002-4250-9724 <muradnz@mcst.ru>*

[1] *АО "МЦСТ", 117437, Москва, ул. Профсоюзная, д. 108.*

[2] *Московский физико-технический институт (национальный исследовательский университет), Россия 117303, Москва, ул. Керченская, д.1 А, корп. 1.*

Аннотация. Работа посвящена развитию смешанных методов анализа неточностей, возникающих при проведении компиляторных оптимизаций. Развитие этих методов важно для процессоров с широким командным словом (VLIW), построенных на архитектуре «Эльбрус» со статическим планированием. Проанализированы существующие подходы к выявлению неточностей в работе оптимизаций, выделены их недостатки. Авторами разработан метод обнаружения неточностей в работе двух важных для VLIW оптимизаций: конвейеризации циклов с аппаратной поддержкой (overlap) и оптимизации выноса участков цикла с малой вероятностью исполнения в создаваемый охватывающий цикл (nesting). Метод реализуется посредством инструментирования циклов в пользовательской программе и получения статической информации о работе циклов от компилятора. Предложенный метод был проверен на задачах из пакетов SPEC CPU 2006 и 2017 rate в режиме base (без использования профильной информации) на ЭВМ с процессором «Эльбрус-8С», где доказал свою эффективность. Метод позволил достичь ускорения до 70.7% на отдельных задачах при расстановке подсказок к оптимизации overlap и 4.71% на задаче 520.omnetpp при расстановке подсказок к оптимизации nesting.

Ключевые слова: компиляторные оптимизации; программная конвейеризация; инструментирование кода; широкое командное слово VLIW.

Для цитирования: Ермолицкий А.В., Левченко Д.Н., Нейман-заде М.И. Определение неточностей в работе некоторых специализированных цикловых оптимизаций в компиляторе LCC для архитектуры «Эльбрус». Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 51–64. DOI: 10.15514/ISPRAS–2025–37(4)–3.

Detection of Inaccuracies in Some Specialized Loop Optimizations in the LCC Compiler for “Elbrus” Architecture

[1] *A.V. Ermolitsky,* ORCID: *0009-0008-2407-7420 <era@mcst.ru>*
[1,2] *D.N. Levchenko,* ORCID: *0009-0004-6744-9889 <levchenko_d@mcst.ru>*
[1,2] *M.I. Neiman-zade,* ORCID: *0000-0002-4250-9724 <muradnz@mcst.ru>*

[1] *AO “MCST”, 108, Profsoyuznaya str., Moscow, 117437, Russia.*

[2] *Moscow Institute of Physics and Technology (National Research University), building 1, 1 A, Kerchenskaya st., Moscow, 117303, Russia.*

Abstract. This work is devoted to the development of mixed methods for analyzing inaccuracies in compiler optimizations. The development of these methods is important for “Elbrus” microprocessors with a very long instruction word (VLIW) because of static scheduling. The article analyzes existing approaches to identifying inaccuracies in optimizations and highlights their disadvantages. In this article a method is developed for detecting inaccuracies in the work of two important for VLIW optimizations: software pipeline with hardware support (overlap) and optimization of moving unlikely executed code into new outer loop (nesting). The method is implemented by instrumenting loops in the user program and obtaining static information about loops from the compiler. The method was checked on SPEC CPU 2006 and 2017 rate suites on a computer with an Elbrus-8S processor and has proven its effectiveness. The method allows to achieve a speedup of 70.7% on test 523.xalancbmk with placing hints for overlap optimization and 4.71% on test 520.omnetpp with placing hints for nesting optimization. Tests are done in base mode without profile information.

Keywords: compiler optimizations; software pipeline; code instrumentation; VLIW.

For citation: Ermolitsky A.V., Levchenko D.N., Neiman-zade M.I. Detection of inaccuracies in specialized loop optimizations in the LCC compiler for “Elbrus” architecture. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025. pp. 51-64 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-3.

1. Введение

Особенность микропроцессоров на основе архитектур со статическим планированием заключается в том, что процесс обнаружения и использования параллелизма инструкций производится только при построении машинного кода оптимизирующим компилятором. Такой подход позволяет существенно упростить микроархитектуру процессора по сравнению с суперскалярными микропроцессорами с внеочередным исполнением, хотя и за счёт определённого усложнения оптимизирующих компиляторов.

Однако такой архитектурный подход обладает своими недостатками. В частности, компилятор (однопроходный) может производить оптимизации только на основе статической информации, используя эвристики, тогда как динамическая информация ему недоступна. Это приводит к тому, что применение компилятором некоторых оптимизирующих преобразований к коду вносит накладные расходы, которые во время исполнения могут оказаться существенными и вместо ускорения пользовательской программы, в таких случаях может происходить её замедление.

Устранить неточности в работе оптимизаций можно с помощью специальных подсказок в исходном коде. Инструмент, рассматриваемый в данной работе, позволяет определить, где и какие именно подсказки следует поставить программисту в исходном коде, чтобы избежать неточностей в работе оптимизаций на горячих участках кода.

Важная особенность рассматриваемого инструмента состоит в том, что необходимо избежать проблем, аналогичных режиму двухпроходной компиляции с использованием профильной информации. В частности, необходимости перекомпиляции и исполнения программы на тестовых данных каждый раз, когда происходит изменение кодовой базы. Поэтому в данном инструменте отчёт о найденных неоптимальностях в работе оптимизаций не подходит для использования в качестве аналога профильной информации.

2. Обзор существующих подходов

Существующие методы в поиске неточностей в работе оптимизаций компилятора делятся на три подхода:

1) Первый подход предполагает использование только динамической информации, полученной при исполнении программы. В данном подходе используются результаты, полученные от инструментов статистического профилирования, таких как perf или dprof. Однако инструменты профилирования программы лишь помогают определить часто исполняемые фрагменты кода, но не говорят о проблемах в работе оптимизаций, так как используют только динамическую информацию и не учитывают статическую информацию от компилятора. При работе с такими профилировщиками нужна высокая квалификация программиста, который должен знать устройство работы оптимизатора в компиляторе и самостоятельно определять проблемы в его работе. Однако программисту придётся столкнуться с рядом проблем. Во-первых, добиться нужного результата невозможно в случае, если используется компилятор с закрытым исходным кодом. Во-вторых, количество оптимизаций со сложными взаимными зависимостями в современных компиляторах исчисляется сотнями. Оптимизации могут создавать контекст для работы друг друга. Разобраться человеку без средств автоматизации оказывается сложно. В-третьих, стоит учитывать проблему доверия профилировщикам. В частности, по своему устройству, статистические профилировщики могут предоставлять лишь статическую информацию о частоте исполнения кода, не являющуюся в полной мере достоверной [1-2]. В свою очередь использование инструментирующих средств профилирования осложнено тем, что они вносят существенные накладные расходы, увеличивая как размеры исполняемых файлов за счёт создания структур в статически выделенной памяти, так и времена исполнения пользовательских программ за счёт дополнительно внедрённого кода.
2) Второй подход опирается только на статическую информацию об исходном коде и информацию от компилятора. Он предполагает использование инструментов обзора примененных оптимизаций. Основным представителем является режим работы opt-report, который поддерживают все современные компиляторы (“llvm-opt-report” у LLVM, “-fopt-report” у LCC, “-qopt-report” у ICC, “-fopt-info” у GCC). Данный режим компиляции показывает программисту какие оптимизации применились и не применились, оперируя только статической информацией без исполнения скомпилированной пользовательской программы. Для анализа проблем в работе оптимизаций, полезным инструментом могут быть программы для обзора бинарных файлов, такие как CCNav [3]. Программа CCNav позволяет сопоставить бинарный файл с исходным кодом вместе с совместным отображением графа потока управления и графа вызовов. Однако данный инструмент, как и в работе с профилировщиками, полагается на знания программиста и не предоставляет никакой информации от компилятора. С учётом того, что «Эльбрус» – это архитектура с поддержкой предикатов, после работы оптимизации слияния альтернатив ветвления кода понять по ассемблерному тексту, какие оптимизации применились к конкретному циклу, очень тяжело. Именно поэтому режим “-fopt-report” является предпочтительным. Общим недостатком этих двух инструментов является отсутствие поддержки динамической информации, поэтому данные инструменты не могут подсказать, перевешивают ли накладные расходы положительный эффект от применения оптимизации или нет.
3) Третий подход является комбинированным подходом, в котором инструменты полагаются и на статическую, и на динамическую информацию. Данный подход предполагает использование комплексных систем анализа производительности, таких как Intel Advisor [4]. Он использует отладочную информацию, которая предоставляется компилятором в режиме “-O2 -g”, информацию от статистического профилирования

и информацию о событиях в процессоре. Благодаря этой информации Intel Adviser может определять проблемы с векторизацией циклов и сообщать пользователю наиболее вероятную причину отсутствия векторизации и способ исправления ситуации. Однако, данный подход лишь косвенно устанавливает причины отсутствия векторизации, но не других оптимизаций, так как это не является актуальным для процессоров с динамическим планированием.

3. Метод поиска неоптимальностей

Предлагаемый в статье метод поиска неоптимальностей относится к третьему подходу. Однако различие с уже известными подходами заключается в том, что в качестве статической информации берётся не отладочная информация о всём исходном коде, а только причины проблем применения оптимизаций к конкретному циклу. Об этих причинах может сообщить компилятор в процессе оценки возможности применения оптимизации. В данной статье рассмотрены способы определения неточностей в оптимизации конвейеризации циклов с аппаратной поддержкой (overlap) [5] и оптимизации выноса участков кода с малой вероятностью исполнения в создаваемый охватывающий цикл (nesting) [6]. Данные оптимизации могут существенно повышать производительность кода циклов при компиляции под архитектуру «Эльбрус».

3.1 Общая схема работы

Анализ неоптимальностей в работе оптимизаций проводится в два этапа.

Первый этап – это этап компиляции. На этом этапе, по указанию, сделанному пользователем, строится дополнительная структура данных, содержащая массивы дескрипторов. Также, в пользовательский код внедряются инструкции увеличения значений определённых счётчиков в указанные дескрипторы. В данной работе под дескриптором будем понимать структуру, которая описывает один цикл. В ней содержится такая информация от компилятора, как:

- привязка к файлу исходного кода;
- предсказанное эвристиками среднее число итераций;
- счётчик заходов в предцикл;
- счётчик количества итераций тела цикла;
- информация о примененных оптимизациях;
- различные служебные флаги для конкретной реализации.

Важно, что вставка дополнительного кода должна вносить как можно меньше побочных эффектов, так как целью является инструментирование работы оптимизаций. Вставка инструментирующего кода может внести возмущения, которые приведут к иному поведению работы оптимизаций. Поэтому инструментирование оптимизаций должно вносить как можно меньше вызовов библиотечных функций и полагаться на вставку линейных участков кода.

При инструментировании циклов, важно помнить, что операции Load и Store могут также вносить возмущения и приводить к иной работе оптимизаций, поэтому, необходимо стараться выносить эти операции за тело инструментируемого цикла. Этого можно достигнуть, если положить объект-счётчик, из соответствующего циклу дескриптора, на регистр перед исполнением тела цикла, а запись в объект сделать после исполнения тела цикла. Соответственно, в теле цикла должна остаться только операция ADD, которая будет вносить минимальное возмущение.

Второй этап – это этап исполнения. Пользователь должен запустить инструментированную раннее программу (без каких-либо дополнительных опций). Стоит отметить, что запуск должен проводиться на репрезентативных данных, отражающих типичные варианты использо-

вания программы. Во время этого исполнения будет набрана статистика за счёт работы операций увеличения значений счётчиков. После окончания работы пользовательской программы вызывается функция библиотеки поддержки, в которой происходит чтение дескрипторов, проведение анализа и вывод отчёта программисту.

3.2 Инструментирование оптимизации overlap

Оптимизация overlap в компиляторе LCC представляет собой программно-аппаратную конвейеризацию циклов, которая позволяет исполнять несколько итераций совместно, как показано на рис. 1б, эффективно используя аппаратные возможности архитектуры «Эльбрус». Благодаря ей, можно эффективно использовать исполняющие устройства в процессоре и увеличить степень заполнения широкой команды. Это одна из самых важных цикловых оптимизаций для архитектуры «Эльбрус», которая позволяет ускорить циклы в несколько раз. Необходимость её инструментирования вызвана тем, что, при малом числе итераций в цикле, накладные расходы могут превышать пользу, которая даёт данная оптимизация. Самые часто замеченные примеры – это неправильное предсказание количества итераций при обходе динамических структур, таких как деревья или списки.

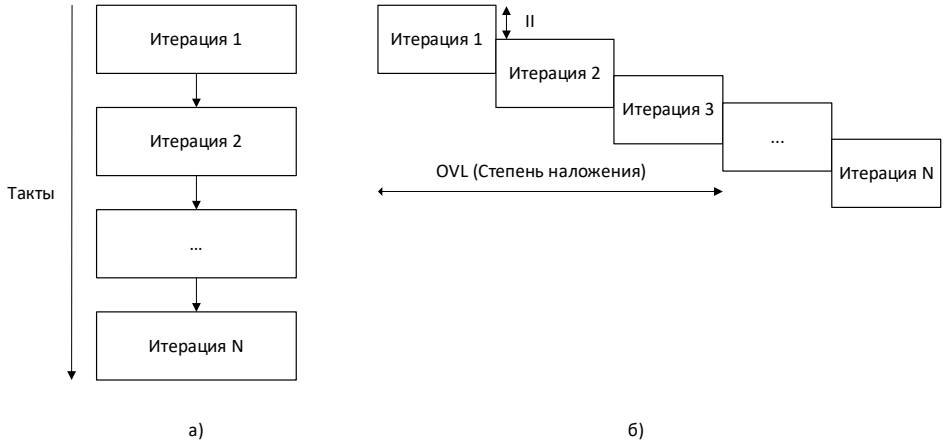

Рис. 1. а) Последовательное исполнение цикла.
б) Исполнение цикла с наложением итераций после оптимизации overlap.
Fig. 1. a) Sequential execution of a loop.
b) Loop execution with overlapping iterations after overlap optimization.

При проведении оптимизации overlap происходит создание дубликата цикла. Для одной из копий цикла производится применение оптимизации. Затем, к каждому из циклов будут применены различные оптимизации. В общей сложности может пройти до 200 фаз оптимизаций вплоть до планирования кода и распределения регистров. На фазе планирования кода подсчитывается время исполнения (с точностью до широкой команды без учёта блокировок конвейера) каждой копии. При штатной работе компилятора, копия цикла с бóльшим оцененным временем исполнения удаляется из кода. Однако для проведения инструментирования необходимо оставить оба цикла, чтобы, когда будет известно точное число итераций вычислить и сравнить время исполнения. Также, необходимо учесть, что оптимизация раскрутки цикла (unroll) может изменить фактическое число итераций.

Важными параметрами, которые необходимо получить от компилятора во время проведения оптимизации overlap являются:

- времена исполнения предцикла, постцикла и тела цикла (для копии, к которой не применилась оптимизация) в тактах без учёта блокировок;
- II (initiation interval) – для копии, к которой применилась оптимизация расстояние

между соседними итерациями (с точностью до широкой команды);

- OVL – для копии, к которой применилась оптимизация, количество одновременно исполняемых итераций.

На основе проведенной подготовки можно вычислить время исполнения обеих копий цикла с точностью до широкой команды без учёта блокировок. Значения $T_{exec(overlapped)}$ и T_{exec} – это времена исполнений цикла, к которым применилась оптимизация overlap и не применилась соответственно.

$$T_{exec(overlapped)} = T_{pre} + T_{post} + II \times (N + OVL - 1) \quad (1)$$

$$T_{exec} = T'_{pre} + T'_{post} + T_{loop} \times N' \quad (2)$$

где N – среднее число итераций цикла, которое может различаться для копий цикла из-за разного фактора раскрутки.

В формуле (1) полезным временем исполнения считается значение $II \times N$. Остальные члены будут считаться накладными расходами. Видно, что чем меньше N, тем большую роль играют накладные расходы. Эмпирически установлено, что при $N < 6$ применять оптимизацию, как правило, невыгодно.

Инструментирование оптимизации overlap производится с целью узнать среднее число итераций. Затем происходит сравнение среднего времени исполнения копии цикла, к которой применилась оптимизация, и времени исполнения копии, к которой оптимизация не применилась. Инструментирование происходит в 3 этапа:

1) На первом этапе от различных фаз компиляции во внутренние структуры компилятора происходит сбор информации о применении к циклам оптимизации overlap. Например, информацию о файле исходного кода можно получить от фазы дублирования циклов, тогда как точное время исполнения предциклов, постцикла и тела цикла можно получить только от фазы планирования кода и распределения регистров.
2) На втором этапе происходит создание дескрипторов каждого инструментированного цикла. В статической памяти пользовательской программы создаётся глобальный массив дескрипторов для каждой функции.
3) На третьем этапе в каждый цикл из функции вставляются операции увеличения значений счётчиков, расположенных в соответствующих дескрипторах циклов.

После исполнения программы выполняется анализ результатов и подготовка отчёта программисту с подсказками, как исправить обнаруженные в ходе анализа неточности в работе оптимизации. Для оптимизации overlap рассматривается три категории проблем:

1) *Циклы, к которым применилась оптимизация overlap*. Для таких циклов рассчитывается время исполнения копий, к которым применилась оптимизация, и к которым она не применилась. Время высчитывается с точностью до широкой команды без учёта блокировок. Времена работы сравниваются, и делается вывод о том, нужно было применять оптимизацию или нет. На рис. 2 приведён пример из отчёта программисту, где применение оптимизации overlap признано нецелесообразным с выдачей подсказки, как можно исправить ситуацию.
2) *Циклы, к которым не применилась оптимизация overlap*. Для таких циклов необходимо определить фактическое количество итераций. Если оно больше 6, то предсказатель профиля в компиляторе допустил ошибку, и необходимо помочь компилятору с помощью подсказки #pragma loop count. Пример такого дескриптора приведён на рис. 3.
3) *Циклы, к которым применилась оптимизация unroll с неоптимальным фактором раскрутки*. Данная оптимизация может уменьшить фактическое число итераций ниже эмпирически установленного предела. На данное поведение в инструменте

также существует правило проверки. В данном случае необходимо выставить подсказку компилятору "#pragma unroll (K_{new})", где K_{new} – наибольший фактор раскрутки, при котором число итераций будет больше 6.

$$6 < \frac{N_{mean} \times K_{old}}{K_{new}}$$

где N_{mean} – среднее число итераций в цикле, K_{old} – фактор раскрутки, который предложил компилятор.

```
src: see.cpp 166-175
preloop counter: 536470178
body counter: 867999569
average iterations: 1.62
predicted iterations by compiler: 10
average loop exec time: 19.00
[------------------------------------ Overlap -------------------------------------]
   Tpre = 7.00, Tpost = 3.00
   II = 3 OVL = 3
   exec time: 19.00
   OVERLAP APPLIED
   Twin loop info:
      Tpre = 0.00, Tpost = 6.00, Tloop = 6.00
      average loop exec time: 12.00
   [SUGGESTION] overlapped loop is slower by more than 25%.
   Use "#pragma noswp" to disable overlap optimization.
[---------------------------------------------------------------------------------]
```

Рис. 2. Пример отчёта оптимизации overlap для одного цикла с большими накладными расходами на применение.
Fig. 2. Example of a loop report for overlap optimization with large overhead.

```
src: ComputeNonbondedBase.h 745-1894
   head counter: 9360
   body counter: 474630
   average iterations: 50.71
   predicted iterations by compiler: 1
   average loop exec time: 52317813.58
   [------------------------------------ Overlap -------------------------------------]
   Tpre = 26.00, Tpost = 0.00, Tloop = 1046355.75
  OVERLAP DIDN'T APPLY
   [SUGGESTION] because of low predicted iterations num: 1
   overlap optimization didn't apply.
   To apply overlap optimization try "#pragma loop count(N)"
   Suggested N: 51
   [---------------------------------------------------------------------------------]
```

Рис. 3. Пример отчёта по оптимизации overlap для одного цикла с неверно предсказанным количеством итераций.
Fig. 3. Example of a loop report for overlap optimization with incorrect predicted loop iteration number.

3.3 Инструментирование оптимизации nesting

Оптимизация nesting позволяет вынести из тела цикла условный код, который имеет малую вероятность исполнения, в специально создаваемый охватывающий цикл. Оптимизация направлена на повышение эффективности конвейеризации цикла, так как при конвейеризации цикла с условным кодом все условные операции будут занимать ресурсы широкой команды, а nesting позволяет их вынести из тела цикла. Оптимизация также может вынести из цикла операции Load/Store, что улучшит работу последующих оптимизаций конвейеризации и позволит спланировать исполнение цикла с меньшим значением II и большей степенью наложения. Пример работы оптимизации приведён на рис. 4.

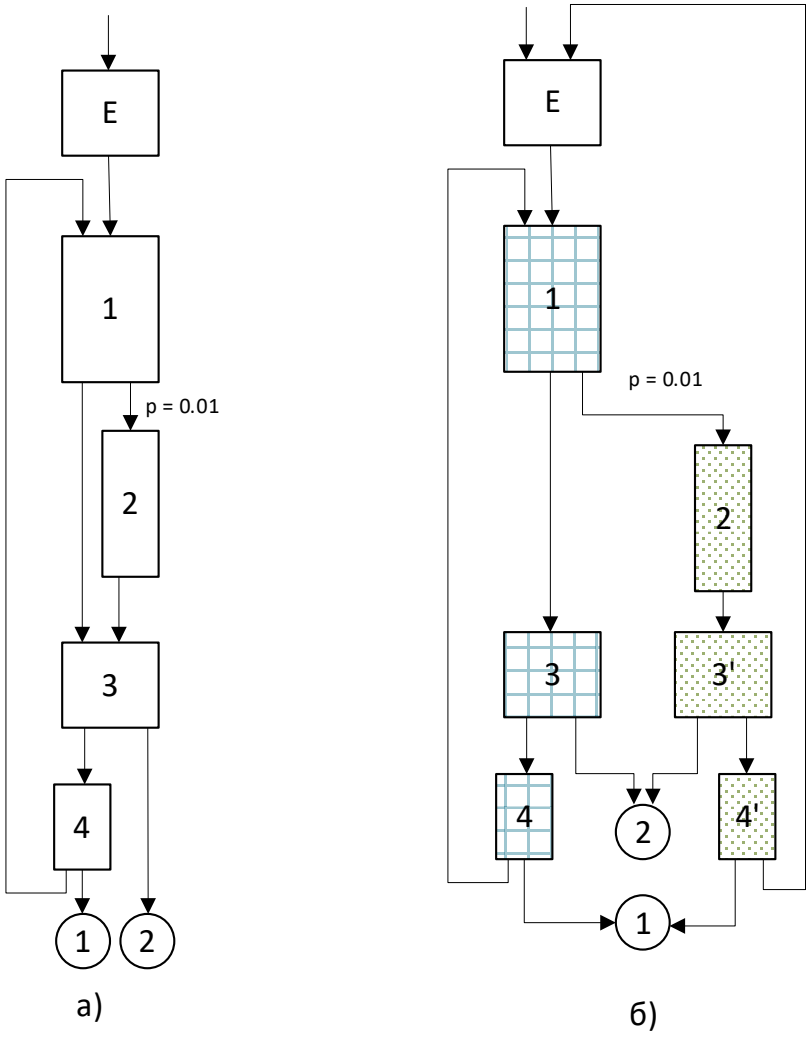

Рис. 4. а) Цикл, имеющий переход с низкой вероятностью. б) Применение оптимизации nesting.
Fig. 4. a) A loop with low branch probability. b) Example of nesting optimization applying.

На рис. 4а приведён пример графа потока управления цикла, в котором имеется условная конструкция ***if***, где одна из альтернатив имеет низкую вероятность перехода. Узел *E* – входной узел для цикла (entry). Узлы 1, 2, 3, 4 образуют цикл. В кругах 1 и 2 показаны номера выходов из цикла. На рис. 4б приведён результат применения этой оптимизации. Оптимизация nesting

производит вынос альтернативы (узел 2) в охватывающий цикл (штриховка точкой), а также дублирует необходимые узлы, чтобы сохранить логику и структуру исполнения программы (узлы 3′ и 4′). У внутреннего цикла (штриховка квадратами), помимо выходов 1 и 2, появляется ещё один выход по дуге с низкой вероятностью. Теперь узел 2 не относится ко внутреннему циклу и оптимизации, которые будут применяться дальше, не будут учитывать переход на данный узел, что позволит спланировать цикл с большей степенью наложения и меньшим значением II.

Для определения частоты исполнения того или иного участка программы, оптимизация nesting существенно полагается на информацию о профиле исполнения программы. Поэтому при однопроходной компиляции оптимизация nesting практически не применяется и считается неэффективной техникой [7].

Целью инструментирования оптимизации nesting является нахождения отношения количества переходов по дуге в охватывающий цикл, которую создала оптимизация nesting, к общему числу итераций тела цикла. Так как охватывающий цикл предварительно считается кодом с низким числом итераций, это число можно проверить на практике, чтобы убедиться, что условие выхода было определено верно, а цикл не является кодом с большим количеством итераций.

На рис. 5 приведён пример инструментирования цикла. Указание *p=0.01* приведено, чтобы выделить дугу графа потока управления, которую построила оптимизация nesting. Инструментирование проводится путём вставки инструкций увеличения значений счётчика итераций цикла (cnt1) и счётчика переходов по дуге в охватывающий цикл, который построен при проведении оптимизации nesting (cnt2).

Отметим, что здесь есть ограничение для дуг, которые ведут на блоки с гарантированным завершением. В случае, если оптимизация nesting посчитала переход на catch-блок маловероятным и вынесла его в охватывающий цикл, то построение инструкций инкрементации счётчиков на такой дуге приведут к разрыву конструкции try-catch и к некорректной работе программы.

После работы инструментирования и исполнения пользовательской программы, получается отчёт с подсказками, которые программист может расставить в коде для помощи компилятору. Пример такой подсказки приведён ниже на рис. 6.

4. Методика измерения производительности

Для измерения ускорения от подсказок инструмента был выбран набор тестов для измерения производительности SPEC CPU [8] 2006 и 2017 rate. Было измерено время исполнения каждого из тестов в составе набора. Затем собраны отчёты по подсказкам к компилятору с помощью предложенного инструмента. В исходном коде нами были внесены поправки в виде подсказок *#pragma {loop count, nonesting, noswp}*. Затем производилась повторная сборка тестов, замеры времени исполнения и сравнение времени исполнения с вариантом без подсказок.

Замеры происходили в базовом (base) режиме. Этот режим предполагает, что все тесты компилируются с одинаковыми опциями, не учитывающие особенности тестов. В данном режиме использовались опции "-O3 -fwhole".

5. Тестовый стенд

В качестве тестового стенда использовалась ЭВМ со следующими характеристиками:

- CPU: Эльбрус 8C, архитектура elbrus-v4, 8 ядер, частота 1.3ГГц.
- RAM: 32Гб
- OS: Эльбрус Линукс 5.4.0-2.3-nn-e8c

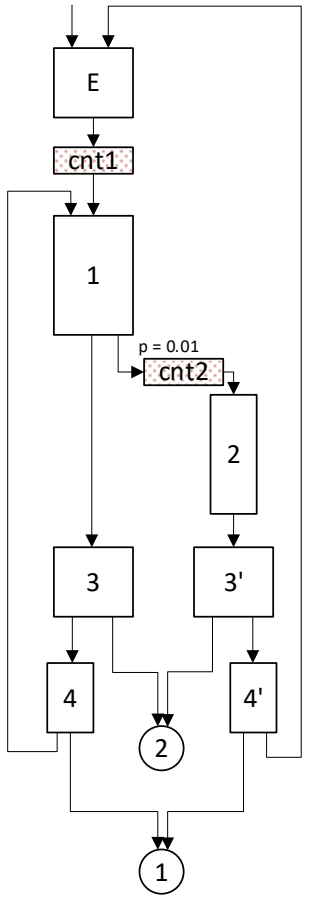

Рис. 5. Инструментирование цикла из рис. 4.
Fig. 5. Loop instrumentation from Fig. 4.

```
Src: physics_types.fppized.f90 547-551
подставлен в physics_types.fppized.f90 582
Nesting counter: 19989504
Body counter: 19989504
Ratio: 100%
[SUGGESTION]: nesting optimization seems unprofitable.
Try to use “#pragma nonesting”
```

Рис. 6. Пример отчёта для одного цикла по оптимизации nesting.
Fig. 6. Example of a loop report for nesting optimization.

6. Замеры производительности

6.1 Оптимизация overlap

На диаграммах рис. 7 и рис. 8 приведены о зафиксированном ускорении исполнения задач из наборов SPEC CPU 2006 и 2017 rate. В эти диаграммы не попали задачи, у которых либо нет подсказок, либо эффект от их расстановки не дал ни ускорения, ни замедления.

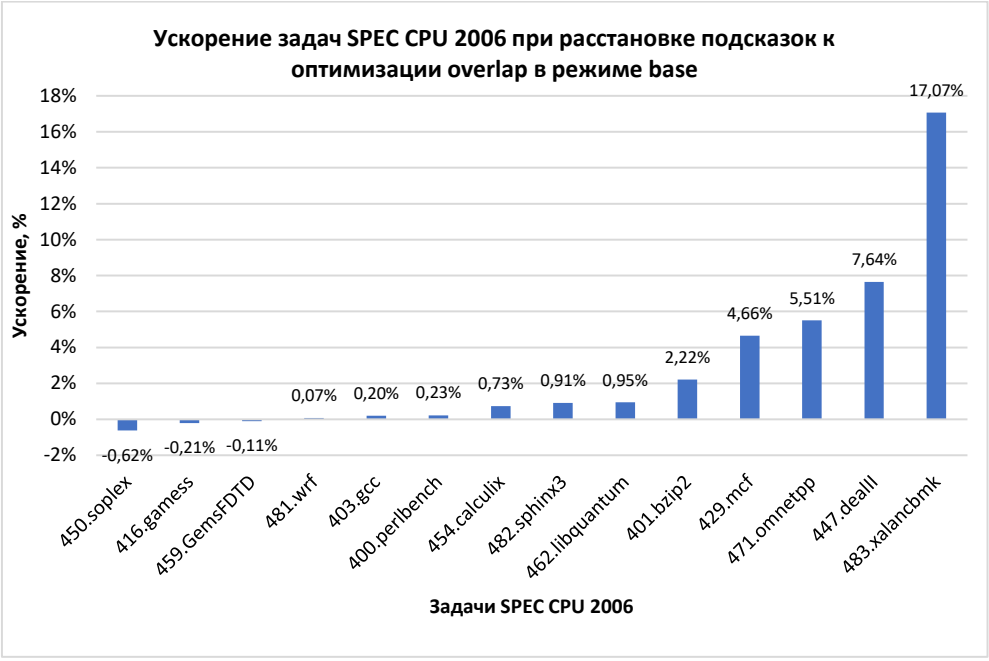

Рис. 7. Ускорение задач SPEC CPU2006
при расстановке подсказок к оптимизации overlap в режиме base.
Fig. 7. Speed up of SPEC CPU 2006 benchmark
with placement of hints to overlap optimization in base mode.

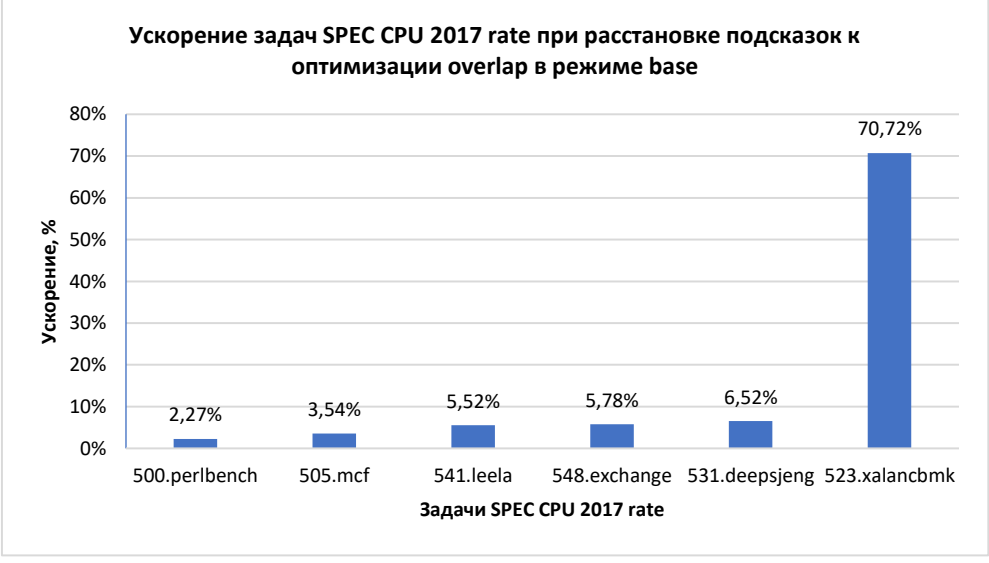

Рис. 8. Ускорение задач SPEC CPU 2017 rate
при расстановке подсказок к оптимизации overlap в режиме base.
Fig. 8. Speed up of SPEC CPU 2017 rate benchmark
with placement of hints to overlap optimization in base mode.

6.2 Оптимизация nesting

В наборе тестов SPEC CPU 2006 обнаружилась только одна задача, где проявился эффект от подсказки “#pragma nonesting”. Это задача *458.sjeng*, ускорение от которой составило **1.43%.**

В наборе тестов SPEC CPU 2017 rate обнаружилась только две задачи: *541.leela* и *520.omnetpp*. Ускорение от расстановки подсказок составило **3.52%** и **4.71%** соответственно.

6.3 Комментарии к результатам

На примере задачи *523.xalancbmk* видно, как могут ошибаться эвристики оценки количества итераций в цикле при однопроходной компиляции. В данной задаче предсказатель профиля компилятора ошибся более, чем в половине циклов во всей задаче. В итоге, расстановка 13 подсказок вида **“#pragma loop count(N)”** позволила ускорить задачу на 70.7%.

Замедление задач *450.soplex*, *416.gamess* и *459.GemsFDTD* обусловлено сложной взаимосвязью между оптимизациями, так как одни оптимизации создают контекст для применения последующих. И даже если применение оптимизации overlap было избыточным, то последующие оптимизации могут всё равно ускорить цикл, тогда как без применения оптимизации overlap для них не появляется контекста.

Малое число обнаруженных проблем в работе оптимизации nesting объясняется тем, что замеры производились в базовом режиме без использования профильной информации. В разделе описания работы оптимизации nesting указано, что это не всегда эффективно и компилятор применяет оптимизацию очень ограниченно.

7. Выводы

В работе рассмотрены существующие методы нахождения неоптимальностей в работе оптимизаций и выявлены их недостатки. Предложен метод нахождения неоптимальностей в работе оптимизаций overlap и nesting в компиляторе LCC, путём инструментирования пользовательской программы. В результате проверки работы метода удалось ускорить часть задач из пакетов SPEC CPU 2006 и 2017 rate. Отметим, что данный метод внедрён в компилятор LCC и будет доступен пользователям начиная с версии 1.31.

Список литературы / References

[1]. Mytkowicz T., Diwan A. et al. Evaluating the Accuracy of Java Profilers, PLDI, 2010, pp. 187-197.
[2]. Hao X., Qingsen W. et al. Can we trust profiling results?: understanding and fixing the inaccuracy in modern profilers. Proceedings of the ACM International Conference on Supercomputing. 2019. pp. 284-295. doi:10.1145/3330345.3330371.
[3]. Devkota S., Aschwanden P. et al. CcNav: Understanding Compiler Optimizations in Binary Code. IEEE Transactions on Visualization and Computer Graphics, 2021, pp. 667-677. doi:10.1109/TVCG.2020.3030357.
[4]. Marques D., Duarte H. et al. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. International Conference in High Performance Computing & Simulation, 2017, pp. 898-907. doi:10.1109/HPCS.2017.150.
[5]. Дроздов А.Ю., Степаненков А.М. Технология оптимизации циклов для архитектур с аппаратной поддержкой конвейеризации // Информационные технологии и вычислительные системы, 2004. №3, с. 52-62.
[6]. Maslennikov D.M., Volkonsky V.Y. Compiler Method and Apparatus for Elimination of Redundant Speculative Computations from Innermost Loops, Pat. US, №6301706B1, 2001.
[7]. Chetverina O. Alternatives of profile-guided code optimizations for one-stage compilation, Programming and Computer Software, 2016, vol. 42, no. 1, pp. 34-40. doi:10.1134/S0361768816010035.
[8]. SPEC Benchmarks and Tools. URL: https://spec.org/benchmarks.html#cpu (дата обращения 02.10.2024).

Информация об авторах / Information about authors

Александр Викторович ЕРМОЛИЦКИЙ – начальник отдела разработки языкового компилятора АО «МЦСТ». Сфера научных интересов: оптимизации в компиляторах.

Alexander Viktorovich ERMOLITSKY – head of the language compiler division at MCST Design Center. Research interests: compiler optimizations.

Дмитрий Николаевич ЛЕВЧЕНКО – старший программист в АО «МЦСТ», аспирант МФТИ. Сфера научных интересов: оптимизации в компиляторах, инструментирующее профилирование, компиляторы для машинного обучения.

Dmitry Nikolaevich LEVCHENKO – software engineer at MCST Design Center and postgraduate student at MIPT. Research interests: compiler optimizations, instrumenting profiling, machine learning compilers.

Мурад Искендер-оглы НЕЙМАН-ЗАДЕ – начальник отделения разработки систем программирования в АО «МЦСТ». Его научные интересы включают методы оптимизации кода, JIT-компиляцию, профилирование и библиотеки ускоренных вычислений.

Murad Iskender-ogly NEIMAN-ZADE – head of the Department of Programming Systems at MCST Design Center. His research interests include compiler optimizations, JIT compilation, profiling and accelerated computation libraries.