DOI: 10.15514/ISPRAS-2024-36(4)-4

Реализация траекторного профилирования в компиляторе LCC для процессоров Эльбрус

В.Е. Шампаров, ORCID: 0000-0001-9900-5159 <shamparov_v@mcst.ru>
М.И. Нейман-заде, ORCID: *0000-0002-4250-9724 <muradnz@mcst.ru>*
АО "МЦСТ", 117437, Москва, ул. Профсоюзная, д. 108.
Московский физико-технический институт
(национальный исследовательский университет),
141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.

Аннотация. В работе предложена реализация траекторного профилирования методом инструментирования, реализованная в компиляторе LCC для архитектур «Эльбрус» и SPARC и предназначенная для улучшения работы специфических оптимизаций для процессоров с архитектурой типа VLIW.

Ключевые слова: компилятор; траекторное профилирование; библиотека поддержки профилирования; широкое командное слово VLIW.

Для цитирования: Шампаров В.Е., Нейман-заде М.И. Реализация траекторного профилирования в компиляторе LCC для процессоров Эльбрус. Труды ИСП РАН, том 36, вып. 4, 2024 г., стр. 41–56. DOI: 10.15514/ISPRAS–2024–36(4)–4.

Path Profiling for LCC Compiler for Elbrus CPUs

V.E. Shamparov, *ORCID: 0000-0001-9900-5159* <shamparov_v@mcst.ru>
M.I. Neiman-zade, ORCID: *0000-0002-4250-9724 <muradnz@mcst.ru>*
AO "MCST", 108, Profsoyuznaya str., Moscow, 117437, Russia.
Moscow Institute of Physics and Technology,
9, per. Institutskij, Dolgoprudnyj, Moskovskaya oblast', 141701, Russia.

Abstract. This paper presents a new version of instrumentation-based path profiling, implemented for the LCC compiler for Elbrus and SPARC processors. This profiling is intended to be used for VLIW-specific compiler optimizations, where path information and path correlations are needed. It was optimized, so profiling overhead descreased to 5.5 times on average.

Keywords: compiler; path profiling; profiling support library; VLIW.

For citation: Shamparov V.E., Neiman-zade M.I. Path profiling for LCC compiler for Elbrus CPUs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 36, issue 4, 2024. pp. 41-56 (in Russian). DOI: 10.15514/ISPRAS-2024-36(4)-4.

1. Введение

Для многих видов оптимизации, применяемых к программам во время компиляции, требуется информация о ходе исполнения программы. Существует множество способов сбора подобной информации, в числе которых находится инструментирование программы компилятором. В этом случае после запуска программы на тренировочных данных компилятору становится доступна информация о ходе исполнения программы.

Классическим является профилирование со сбором счётчиков дуг графа потока управления. Но данный способ не даёт информации про корреляцию между путями исполнения программы, которая может понадобиться для некоторых оптимизаций. Для решения данного вопроса применяются траекторные профили, в которых собрана статистика траекторий, которые проходила программа.

В данной работе предложена реализация траекторного профилирования, предназначенная для улучшения работы специфических оптимизаций для процессоров с архитектурой типа Very Long Instruction Word (VLIW). Работа состоит из нескольких частей. В разделе 2 приведена решаемая проблема. Связанные с темой данной работы публикации описаны в разделе 3. Созданные для решения проблемы алгоритмы описаны в разделе 4. В разделе 5 приведены полученные результаты. Наконец, в разделе 6 подведены итоги совершённой работы и описаны планы дальнейшей работы над траекторным профилированием.

2. Недостаточность информации классического профиля

Для архитектуры «Эльбрус», являющейся архитектурой типа VLIW с явным параллелизмом исполнения команд и имеющей предикатный режим исполнения команд, существенно важна оптимизация слияния кода (она же if-конверсия, она же оптимизация слияния альтернатив условий [15]).

Алгоритм оптимизации слияния кода предполагает следующие действия:

1. выбор региона – набора узлов графа потока управления для слияния; среди выбранных узлов есть голова региона, и все входы в регион должны быть входами в голову;
2. дублирование узлов с внешними входами в регион (не в голову) для обеспечения требования в предыдущем пункте;
3. слияние региона в один гиперузел с использованием предикатного режима исполнения кода; таким образом получается, что в гиперузле на исполнение поступают все слитые пути, а по предикатам происходит выборка операций, которые действительно начинают исполняться;
4. спекулятивный вынос части операций вверх из-под предикатов.

Из-за использования предикатного режима при исполнении на процессорах с последовательным исполнением команд (in-order), например, процессорах «Эльбрус», операции под отрицательным предикатом занимают место на устройствах исполнения команд, но не исполняются. Из-за этого существенным требованием является отсутствие в регионе маловероятных путей исполнения, так как они с высокой вероятностью бесполезно занимают место на устройствах и при этом негативно влияют на планирование кода.

Для качественного определения маловероятных узлов графа потока управления используются результаты проведённого профилирования либо предсказанный профиль. Но правильная коррекция профиля при дублировании узлов данным способом недостижима, так как требуется корреляция между траекториями до дублируемого узла и траекториями после него. Без правильной коррекции профиля при дублировании невозможно достичь правильного выбора регионов, так как нельзя корректно определить маловероятные узлы на траекториях после продублированного узла.

На рис. 1 изображён пример участка графа потока управления, где классический профиль по счётчикам дуг не всегда позволяет оптимальным образом провести слияние узлов. Пусть из-за зависимостей в коде получается так, что после слияния кода операции из узлов, связанных дугами до слияния, не будут смешиваться, а длительность узлов следующая:

- *A* – 10 тактов;
- *B* – 5 тактов;
- *C* – 15 тактов;
- *D* – 10 тактов;
- *E* – 15 тактов;
- *F* – 5 тактов;
- *G* – 10 тактов.

Также пусть любая пара узлов, которые по данным не зависимы друг от друга, при планировании параллельно друг другу не увеличивают свою длительность.

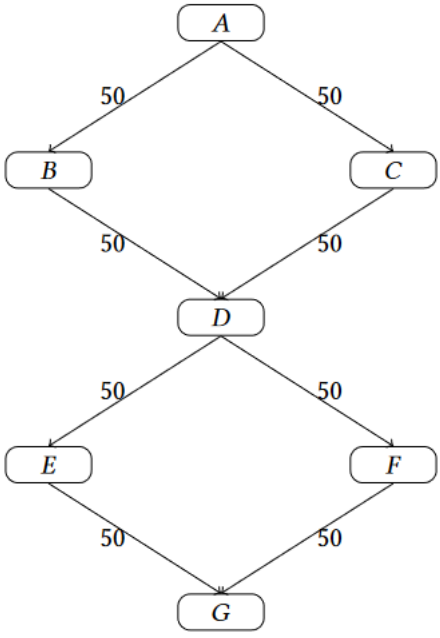

Рис. 1. Пример участка графа потока управления, где классический профиль по счётчикам дуг не всегда позволяет оптимальным образом провести слияние узлов.
Дуги маркированы своими счётчиками.
Fig. 1. Example of CFG part, where classic arcs profile does not always allow optimal if-conversion.
Edges are marked with their counters.

В ситуации, когда при исполнении выполняются только траектории *ABDEG* и *ACDFG* по 50 раз каждая, они формируют счётчики дуг, как показано на рисунке. В этом случае без информации о траекториях компилятор производит слияние кода в гиперузел как на рис. 2. Прерывистые дуги на рисунке обозначают зависимость по данным. Узел *A* генерирует предикат p_A, узел *D* – предикат p_D. Код узла *B* выполняется при условии истинности предиката p_A, код узла *C* выполняется при условии ложности предиката p_A. Аналогично и узлы *E* и *F* относительно предиката p_D. Длительность такого гиперузла составляет 60 тактов.

При наличии информации о траекториях оптимальное слияние в гиперузел изображено на рис. 3. В этом случае узел D и последующие узлы E, F и G дублируются, но маловероятные пути среди дублированных узлов выносятся из гиперузла в отдельные «холодные» гипе-рузлы. Длительность основного («горячего») гиперузла составляет уже 50 тактов.

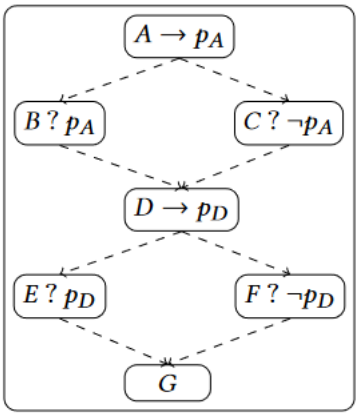

Рис. 2. Пример участка графа потока управления с рис. 1, слитого без учёта траекторий.
Fig. 2. Example of CFG part from fig. 1 after if-conversion without paths.

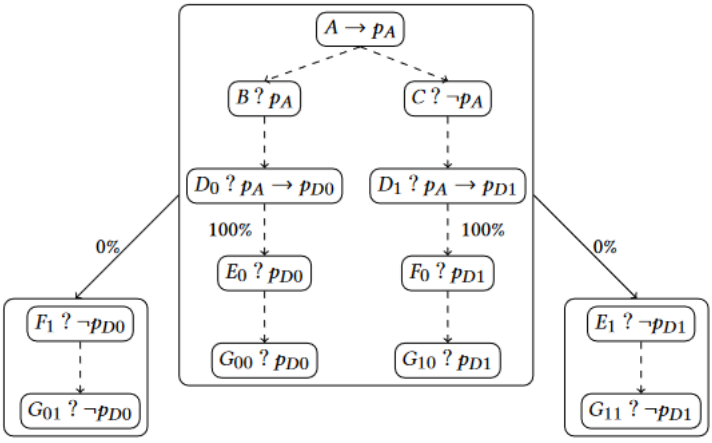

Рис. 3. Пример участка графа потока управления с рис. 1, слитого с учётом траекторий.
На дугах из дублированного узла D отображены вероятности переходов.
Fig. 3. Example of CFG part from fig. 1 after if-conversion with paths.
Edges from duplicated node D are marked with probabilities for control transfer towards them.

Таким образом, для коррекции профиля при дублировании узлов в ходе оптимизации слияния кода требуется траекторный профиль с информацией о корреляциях между траекториями до дублируемых узлов и траекториями после дублируемых узлов.

Также для иных цикловых оптимизаций требуется информация о корреляции траекторий в разных итерациях цикла, а для оптимизации inline – траектории, проходящие через операции `call` и `return`.

3. Работы по данной тематике

Известна работа [3], в которой предложен легковесный алгоритм сбора траекторий с присвоением каждой нециклической траектории в функции или в цикле уникального номера. Указанный алгоритм делит всю программу на такие ациклические участки по границам циклов и функций и позволяет получить счётчик каждой такой траектории. За счёт алгоритма нумерации траекторий номер траектории к концу ациклического участка получается из серии инкрементов счётчика на определённых пройденных дугах. Соответственно, на большинстве дуг программы либо не создаётся операций, либо создаётся единственная операция инкремента счётчика. И только на границах ациклических участков и функций добавляется код инкремента счётчика исполнений полученной траектории.

Легковесность приведённого алгоритма является его преимуществом, но из-за разделения кода на ациклические участки данный способ профилирования не позволяет использование для межпроцедурных и некоторых цикловых оптимизаций.

Для траекторного профилирования методом, описанным в [3], было сделано несколько улучшений. В работе [12] за счёт изменения алгоритма нумерации траекторий удалось ещё сильнее уменьшить накладные расходы на профилирование – в улучшенном алгоритме инкременты счётчиков оказываются на более редких траекториях.

Интересное улучшение было предложено в работе [4] для систем, в которых работающая программа находится под наблюдением работающего в другом потоке профилировщика либо иных подобных профилировщику программ. Например, в этой статье используется JIT-компилятор, занимающийся профилированием программы. В этом случае вместо увеличения счётчика для конкретного пути, что авторы считают дорогостоящей операцией, профилировщик по сигналу от программы в месте записи получает из программы номер пути и далее уже сам записывает полученную информацию. Таким образом, накладные расходы на запись номера траектории перекладываются на профилировщик.

Ещё один способ уменьшения накладных расходов на профилирование предложен в работе [2] для многоядерных систем, где предлагается выполнять несколько экземпляров профилируемой программы параллельно, и в каждой из них инструментировать разные части. Таким образом, накладные расходы на профилирование разделяются между разными экземплярами.

В работе [8] предложен способ обобщения траекторного профилирования для сбора в том числе межпроцедурных траекторий, но для этого строится суперграф (то есть граф потока управления для всех функций сразу с дугами по вызовам функций), а на дугах строятся более сложные вычисления, чем инкременты счётчиков. Этот профиль может быть использован для оптимизации частичной подстановки кода вызываемых функций. Но при инструментировании больших программ может не хватать оперативной памяти, так как строится суперграф.

В работе [13] предложена идея разделять процедуру на иерархию вложенных подграфов и профилировать траектории в подграфах независимо. Это позволяет в JIT-компиляторах гибко собирать только нужную информацию и в разумное время принимать решения о перекомпиляции участков кода на основании таких частичных траекторных профилей.

В работе [11] сделан подход к сбору траекторий разных итераций циклов (максимальной длиной k ациклических подтраекторий, по одной на итерацию) для определения корреляций между ними. В данной работе для каждого цикла предложено собирать собственные коллекции траекторий длиной k итераций. Но полноценно данный подход удалось развить в работе [6]. Там предложен более общий алгоритм, позволяющий для каждой функции собирать коллекции траекторий длиной k ациклических подтраекторий. Оба подхода позволяют выявить корреляции между разными итерациями циклов на расстоянии до $k - 1$ итераций.

Иной вид профиля предложен в статье [14]. Там авторы поставили цель улучшить расположение базовых блоков программы, в том числе с помощью их дублирования, для улучшения работы предсказателя переходов. Для этого они собирают профиль траекторий, подобный собираемому в предсказателе переходов [9]. Траектория состоит из данных для k предыдущих

операций `branch` перед текущей, в которых сохраняется номер базового блока, в котором была операция, и результат: произошёл или не произошёл переход. Для каждой такой траектории собираются два счётчика: сколько раз история произошла и сколько раз из них `branch` привёл к переходу (branch was taken). На основе этих данных, собранных в дерево для каждой операции `branch`, для каждого узла с операцией `branch` производится предсказание, какие варианты там возможны: переход происходит, не происходит или возможны оба варианта. Узлы, для которых предсказатель может давать оба варианта, дублируются так, чтобы предсказатель переходов редко делал неверные предсказания.

Ещё один вид траекторного профиля, предложенный в работе [10], требует широкой поддержки со стороны аппаратуры. В данной работе за основу профиля взята комбинация из битового вектора результатов операций `branch` на данном ациклическом участке кода и полученного аппаратно счётчика времени в тактах, которое заняло конкретное исполнение участка кода. На основе таких данных определяются вариации времени исполнения каждой траектории в программе. Если время выполнения конкретных траекторий конкретного участка кода сильно вариативно, то можно считать, что этот участок кода недооптимизирован. В статье не предложено способов автоматического использования результатов профилирования, кроме приоритизации участков кода для оптимизации.

Поддержка собираемого аппаратно траекторного профиля предложена компанией Intel и используется, например, в утилите BOLT [9]. Аппаратура собирает последние k произошедших переходов в формате (откуда – куда) в специальный буфер (регистр) LBR, и это позволяет программе или профилировщику получить эти данные и, переведя в вид узлов CFG, получить траектории в программе. В утилите BOLT данные возможности используются для расположения базовых блоков программы в оптимальном порядке.

Для архитектур без LBR предложен способ собирать трассы в том же формате в работе [7], хотя в целях уменьшения накладных расходов это сделано в формате сэмплирования. Во время остановки для сбора сэмпла сбор одной трассы длиной k происходит в несколько этапов. Сначала, начиная с адреса текущей команды, происходит поиск k последующих операций перехода. Если операция безусловная, то она декодируется и соответствующая запись добавляется в трассу. Если же операция находится под условием, то на неё ставится точка останова, и программа продолжает исполнение. Вскоре программа будет остановлена на этой точке останова, и тогда появится возможность выяснить, произойдёт там переход либо нет. В первом случае в трассу добавляется соответствующая запись. В обоих случаях можно продолжить собирать трассу аналогичным способом, пока в трассе не наберутся k записей.

4. Реализованное траекторное профилирование

В рамках данной работы реализовано инструментирование в оптимизирующем компиляторе LCC для процессорных архитектур «Эльбрус» и SPARC и библиотека поддержки для сбора профильных данных.

Вид собираемого профиля выбран на основе работы [14], но с существенными отличиями:

- так как в общем случае у узла графа потока управления может быть более 2 исходящих дуг, однобитный флаг произошедших или не произошедших переходов не подходит, поэтому для идентификации исходящей дуги выбран номер выбранной исходящей дуги;
- в ходе оптимизации слияния кода происходит дублирование узлов с множественными входными дугами, соответственно для коррекции профиля после дублирования потребуется различать траектории для разных входных дуг; из-за этого помимо узлов с множественными выходными дугами происходит сбор узлов с множественными входными дугами;

- для коррекции профиля нужна история после узла с множественными входными дугами, поэтому сохраняемые траектории имеют следующий вид:
- *n* узлов с множественными выходными дугами (чаще всего с операциями `branch`), для каждого указана конкретная исполненная дуга; будем называть эту часть *«предшествующей траекторией»*;
- узел с множественными входными дугами;
- *k* узлов с множественными выходными дугами, для каждого указана конкретная исполненная дуга; будем называть эту часть *«последующей траекторией»*.

Назовём *«узлами схождения»* (также просто *«схождениями»*) узлы с множественными входными дугами и исполненной конкретной дугой среди данных входных, а *«узлами расхождения»* (также просто *«расхождениями»*) − узлы с множественными выходными дугами и исполненной конкретной дугой среди данных выходных. Таким образом, сохраняются траектории из *n* расхождений до схождения, самого схождения и *k* расхождений после него.

В целях возможного будущего применения данного профиля для межпроцедурных оптимизаций, например, частичной подстановки кода вызываемых функций как в работе [8], на собираемые профили не накладывается никаких ограничений на вхождение составляющих их узлов графа потока управления в разные процедуры. Из-за этого для узлов разных функций строятся уникальные идентификаторы для однозначного определения узлов. Идентификатор узла с дугой (входящей либо исходящей) состоит из следующих частей:

- идентификатор единицы трансляции;
- идентификатор функции в единице трансляции;
- уникальный номер узла внутри графа потока управления функции (выбран номер по RPO-нумерации);
- уникальный номер дуги.

Созданные во время компиляции идентификаторы единиц трансляции и функций записываются в специальный файл метаданных для использования во время применения собранного траекторного профиля.

4.1 Неоптимизированное инструментирование

Алгоритм инструментирования во время компиляции одной единицы трансляции состоит из следующих этапов:

1) для каждой функции в единице трансляции сгенерировать уникальный идентификатор; для этого подходит простое назначение каждой новой функции последовательных целых неотрицательных чисел;
2) для каждой функции:
    1. сбор узлов схождения и расхождения с запоминанием их идентификаторов; это делается для того, чтобы созданные во время инструментирования узлы и дуги не влияли на собираемые в профиле данные;
    2. непосредственно инструментирование каждого узла схождения и расхождения.

В обычном случае инструментирование узла расхождения предполагает вставку на каждой исходящей дуге вызова функции `TraceDivNode` из библиотеки поддержки, куда подаются идентификаторы единицы трансляции, функции, узла и данной исходящей дуги. Аналогично производится инструментирование узла схождения − вызовы `TraceConvNode` с параметрами в виде идентификаторов единицы трансляции, функции, узла и данной исходящей дуги вставляются на каждой входящей дуге.

Исключение – случаи, когда на дуге нельзя вставлять никакого кода. Например, такое происходит на дугах, соединяющих узел с потенциально вызывающим исключения вызовом с узлом обработчика исключения. В таких случаях требуется специальная версия инструментирования. Для узлов расхождения инструментирование выглядит следующим образом:

1) в узел с множественными выходами добавляется запись идентификатора узла в специально созданную временную переменную;
2) вызов `TraceDivNode`, где в параметрах находится значение временной переменной, вставляется ниже по графу потока управления от дуги настолько рано, насколько это возможно.

Аналогично и для узлов схождения:

1) в узел перед дугой вставляется запись идентификатора дуги в специально созданную временную переменную;
2) вызов `TraceConvNode`, где в параметрах находится значение временной переменной, вставляется ниже по графу потока управления от дуги настолько рано, насколько это возможно.

Таким образом, на каждой входной дуге узлов схождения и на каждой выходной дуге узлов расхождения (либо рядом с ними) оказывается по вызову функции из библиотеки поддержки.

4.2 Неоптимизированная библиотека поддержки

В библиотеке поддержки находятся два буфера:

- буфер узлов расхождения;
- буфер узлов схождения.

Для каждого узла схождения хранится индекс следующего во времени узла расхождения для того, чтобы корректно определять нужные траектории.

Функции `TraceConvNode` и `TraceDivNode` заполняют эти буферы до тех пор, пока один из буферов не заполнится. Затем производится частичное заполнение статистики, при котором происходит обход буферов, построение всех находящихся там накопленных траекторий и инкременты счётчиков для каждой из них.

Подобный способ промежуточного сохранения информации для сбора траекторий позволяет иметь два буфера на всю профилируемую программу и собирать траектории, включающие схождения и расхождения из разныю функций и единиц трансляции.

Статистика хранится в виде вложенных хэш-таблиц:

- ключ первого уровня – схождение; таким образом, вложенная таблица имеет смысл статистики для данного схождения; например, для примера на рис. 1 возможные ключи:
  - узел *D* с дугой *BD*;
  - узел *D* с дугой *CD*;
  - узел *G* с дугой *EG*;
  - узел *G* с дугой *FG*;
- ключ второго уровня – предшествующая траектория; таким образом, вложенная таблица имеет смысл распределения счётчиков для данной предшествующей траектории; например, для примера на рис. 1 и схождения в виде узла *D* с дугой *BD* возможный ключ длины 1 единственен − узел *A* с дугой *AB*;
- ключ третьего уровня – последующая траектория; значение − счётчик; например, для примера на рис. 1 для схождения в виде узла *D* с дугой *BD* и предшествующей

траектории из одного расхождения в виде узла *D* с дугой *BD* возможно следующее состояние таблицы:

- ключ – расхождение в виде узла *D* с дугой *DE*, значение – счётчик 10;
- ключ – расхождение в виде узла *D* с дугой *DF*, значение – счётчик 40.

Соответственно, для очередной траектории для инкремента счётчика достаточно найти или создать его в трёхуровневой таблице.

Подобное представление статистики позволяет легко получить условные вероятности последующих траекторий в зависимости от предшествующих. Например, для указанного для третьего уровня таблицы примера условные вероятности исполнения дуг *DE* и *DF* при условии произошедшей предшествующей траектории из одного расхождения в виде узла *D* с дугой *BD* и схождения в виде узла *D* с дугой *BD* составляют:

- дуги *DE* – 20%;
- дуги *DF* – 80%.

Из-за буферизации могут быть потеряны несколько траекторий на краях буфера схождений или расхождений. Доля потерянных траекторий тем меньше, чем больше размер буферов. Также из соображений оптимальности работы кэша инструкций и цикловых оптимизаций для цикла обработки буферов предпочтительны большие буферы. Выбраны размеры буферов по 2^{25} элементов: они достаточно большие и в сумме занимают в памяти процесса около 0,5 Гб, что является приемлемой нагрузкой на память программы.

4.3 Оптимизация

Исследование накладных расходов на профилирование неоптимизированных версий инструментирования и обработки показало, что наибольшее замедление достигается из-за вызова функций библиотеки поддержки на каждой инструментированной дуге, тогда как внутри эти функции в обычном случае только пополняют соответствующие буферы. Соответственно, для ускорения работы требуется как можно сильнее уменьшить количество вызовов библиотеки поддержки.

Для этого в самой программе при инструментировании создаются буферы узлов схождения и расхождения. Но для того, чтобы в программе оказалось только по одному буферу каждого вида, программа должна собираться в режиме `-fwhole` или `-flto`.

На инструментируемых дугах вместо вызовов библиотеки поддержки строится специальный код для пополнения буферов и редкого вызова функции из библиотеки поддержки для обработки буферов и пополнения статистики. Эквивалентный код инструментирования на языке Си представлен на рис. 4 и рис. 5. В этих примерах буферы схождений и расхождений `conv_buffer` размером `max_conv_size` и `div_buffer` размером `max_div_size`. Также в данных примерах счётчики актуальных размеров буферов – `conv_size` и `div_size`.

Таким образом получается, что вместо вызовов функций библиотеки поддержки на каждой инструментируемой дуге вызовы происходят в среднем 1 раз на `max_conv_size` или `max_div_size` прохождений инструментируемых дуг за счёт переноса буферов схождений и расхождений из библиотеки поддержки в профилируемую программу.

Помимо этого, имеются и оптимизации в самой библиотеке поддержки:

- эксперименты показали, что количество частых траекторий до и после узла схождения мало; таким образом, хэш-таблицы второго и третьего уровня можно заменить на TNV-таблицы [5] – массивы фиксированного размера из ключей, значений и счётчиков, где при увеличении счётчика элемент может "всплыть" на одну позицию выше; при использовании периодической очистки нижней (более редкой) половины такой таблицы и при условии, что количество частых элементов не более

половины размера TNV-таблицы, вероятность упустить какой-то частый элемент становится крайне малой;

```
if ( conv_size >= max_conv_size ) {
    TraceBunch( conv_buf,
                conv_size,
                div_buf,
                div_size );
    conv_size = 0;
    div_size = 0;
}
conv_node.module = module_id;
conv_node.func = func_id;
conv_node.node = node_id;
conv_node.edge = edge_id;
conv_buf[conv_size].node     = conv_node;
conv_buf[conv_size].next_div = div_size;
conv_size++;
```

Рис. 4. Код инструментирования, создаваемый на дуге узла схождения.
Fig. 4. Instrumentation code, placed at every edge before CFG node with multiple predececcor edges.

```
if ( div_size >= max_div_size ) {
    TraceBunch( conv_buf,
                conv_size,
                div_buf,
                div_size );
    conv_size = 0;
    div_size = 0;
}
div_node.module = module_id;
div_node.func = func_id;
div_node.node = node_id;
div_node.edge = edge_id;
div_buf[div_size] = div_node;
div_size++;
```

Рис. 5. Код инструментирования, создаваемый на дуге узла расхождения.
Fig. 5. Instrumentation code, placed at every edge after CFG node with multiple successor edges.

- на первом уровне таблицы введена регулярная фильтрация редких узлов схождения таким образом, чтобы за всё время профилирования суммарный счётчик удалённых траекторий составил около 1%;
- наивная реализация добавления траектории предполагает активное создание копий данных, хотя они продолжают храниться в буферах; аккуратное использование передачи указателей (с копированием данных только в ключи таблицы) и раскладки ключей в TNV-таблицах в памяти подряд позволило снизить потери на аллокации, деаллокации и кэш-промахи до приемлемых;
- добавлена возможность разрежения получаемого профиля с фактором N: обрабатывается только один набор из N буферов; также добавлена возможность разрежения с фактором, растущим по простым числам;
- обработка буфера узлов расхождения сделана параллельной: каждый поток исполнения (всего их количество равно числу ядер процессора) получает задание на

обработку своей части буфера и хранит свою частичную статистику, при завершении работы программы статистики объединяются.

5. Результаты

Предложен и реализован вариант траекторного профиля, подходящий для коррекции профиля после дублирования узлов при слиянии кода.

Характеристики реализованного профилирования были измерены на компьютере с процессором «Эльбрус-8СВ» (архитектура «Эльбрус» типа VLIW [16], 8 ядер, частота 1,55 ГГц) на бенчмарках SPEC CPU1995, SPEC CPU2000 и SPEC CPU2006 [1]. Характеристики профилирования для измерений:

- фактор прореживания − растущий по простым числам;
- длина траекторий − 4 узла до и 2 после;
- размеры TNV-таблиц для траекторий до и после − 32 элемента для предшествующей траектории и 16 элементов для последующей траектории.

В табл. 1-3 показаны замедления задач наборов SPEC CPU1995, 2000 и 2006 и доля траекторий, чьи счётчики остались после фильтраций, для каждой задачи.

Получены следующие усреднённые характеристики профилирования:

- замедление из-за профилирования:
  - среднее геометрическое: 5,5 раз.
  - медианное: 6,3 раза.
  - худшее: 90,9 раз.
  - лучшее: на 2%.

Табл. 1. Замедление и доля траекторий, чьи счётчики остались после фильтраций, для задач бенчмарка SPEC CPU1995.
Table 1. Slowdown and saved path ratio (after all filtrations) for SPEC CPU1995 benchmark.

Задача	Доля сохранённых траекторий	Замедление, раз
099.go	97,01%	9,174
101.tomcatv	99,73%	1,250
103.su2cor	99,06%	2,632
104.hydro2d	99,50%	41,667
107.mgrid	98,92%	2,128
110.applu	99,99%	6,329
124.m88ksim	98,98%	1,894
125.turb3d	99,01%	3,096
126.gcc	95,63%	6,250
129.compress	99,92%	5,682
130.li	95,77%	10,204
132.ijpeg	98,82%	12,346
134.perl	99,41%	9,259
141.apsi	99,37%	20,000
145.fpppp	99,38%	12,658

146.wave5	99,59%	7,937
147.vortex	96,91%	5,525

Табл. 2. Замедление и доля траекторий, чьи счётчики остались после фильтраций, для задач бенчмарка SPEC CPU2000.

Table 2. Slowdown and saved path ratio (after all filtrations) for SPEC CPU2000 benchmark.

Задача	Доля сохранённых траекторий	Замедление, раз
164.gzip	98,25%	9,615
168.wupwise	99,73%	1,020
172.mgrid	99,76%	1,953
173.applu	100,00%	7,042
175.vpr	99,02%	3,610
176.gcc	96,96%	1,548
177.mesa	98,80%	1,439
178.galgel	98,81%	2,755
179.art	99,93%	1,292
181.mcf	99,10%	2,639
183.equake	99,89%	1,792
186.crafty	93,95%	4,651
187.facerec	99,19%	3,472
189.lucas	99,02%	8,197
191.fma3d	98,99%	6,024
197.parser	90,17%	10,101
200.sixtrack	98,99%	7,937
252.eon	99,58%	8,621
253.perlbmk	99,58%	3,906
254.gap	93,23%	13,333
255.vortex	97,62%	9,709
256.bzip2	98,25%	7,407
300.twolf	98,93%	7,752
301.apsi	99,11%	8,929

- доля траекторий, чьи счётчики остались после всех фильтраций:
  - среднее геометрическое: 97,8%.
  - медианное: 98,9%.
  - худшее: 82,1%.
  - лучшее: 100,0%.
- доля предшествующих траекторий, которые показывают существенную разницу в распределениях вероятности исходящих дуг ближайшего узла расхождения после:
  - среднее геометрическое: 9,6%.
  - медианное: 7,9%.
  - худшее: 1,6%.

- лучшее: 98,5%.

Анализ одного из худших случаев по замедлению (задачи 104.hydro2d, замедляющейся в 42 раза) показал, что значительную долю в замедление вносит сам инструментированный код: задача без библиотеки поддержки замедлилась примерно в 11 раз, а библиотека вносит ухудшение всего около 4 раз. Причина – в том, что многие цикловые оптимизации перестают работать при наличии в теле цикла операций вызова (пусть даже и редких), и иногда добавленное сложное управление (условный оператор с редким переходом по истинности условия) тоже мешает проведению цикловых оптимизаций.

Табл. 3. Замедление и доля траекторий, чьи счётчики остались после фильтраций, для задач бенчмарка SPEC CPU2006.

Table 3. Slowdown and saved path ratio (after all filtrations) for SPEC CPU2006 benchmark.

Задача	**Доля сохранённых траекторий**	**Замедление, раз**
400.perlbench	98,25%	7,092
401.bzip2	96,98%	5,747
403.gcc	95,98%	9,091
410.bwaves	99,91%	11,494
416.gamess	97,51%	1,188
429.mcf	98,39%	4,115
433.milc	99,16%	3,937
434.zeusmp	99,61%	1,661
435.gromacs	97,90%	9,009
436.cactusADM	99,02%	6,452
437.leslie3d	98,95%	5,952
444.namd	99,09%	6,061
445.gobmk	94,08%	2,967
447.dealII	91,21%	2,597
450.soplex	98,94%	3,356
454.calculix	98,19%	3,165
456.hmmer	99,11%	1,484
458.sjeng	91,15%	5,181
459.GemsFDTD	98,64%	2,994
462.libquantum	99,00%	6,897
464.h264ref	97,77%	50,000
465.tonto	82,08%	90,909
470.lbm	99,99%	5,917
471.omnetpp	98,01%	2,801
481.wrf	98,34%	7,519
482.sphinx3	97,95%	6,897
483.xalancbmk	97,06%	10,638

6. Заключение и направления развития

Разработанный вариант траекторного профилирования имеет приемлемые характеристики по накладным расходам, хотя в некоторых случаях имеет большое замедление, вызванное выключением работы оптимизаций из-за инструментирования. В дальнейшем (как развитие данной работы) предполагается вести исследования в следующих направлениях:

- применение профиля к оптимизации слияния кода;
- использование аппаратных возможностей процессоров «Эльбрус», схожих с LBR, для получения траекторий аппаратным способом; для этого потребуется приводить собранные трассы к описанному в данной статье виду;
- использование профилей по [6], представляющих определённый интерес в плане легковесности профилирования, для генерации профилей описанного в данной статье вида; для этого потребуется приводить ациклические номера траекторий к траекториям из узлов схождения и расхождения и проводить дополнительный анализ.

Для реализованной библиотеки поддержки траекторного профилирования получено свидетельство о государственной регистрации программы для ЭВМ №2023685378.

Список литературы / References

[1]. SPEC CPU Benchmarks, Available at: https://www.spec.org/benchmarks.html#cpu, accessed 23.05.2024.
[2]. Mohammed Afraz, Diptikalyan Saha, and Aditya Kanade (2015) P3: Partitioned Path Profiling. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 485–495. DOI: 10.1145/2786805.2786868.
[3]. Thomas Ball and James R. Larus (1996) Efficient Path Profiling. In Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29). IEEE Computer Society, USA, 46–57.
[4]. M.D. Bond and K.S. McKinley (2005) Continuous path and edge profiling. DOI: 10.1109/MICRO.2005.16.
[5]. Eustace A. Calder B., Feller P. Value Profiling and Optimization. Journal of Instruction-Level Parallelism 1 (1999), pp. 1–37.
[6]. Daniele Cono D'Elia and Camil Demetrescu. Ball-Larus Path Profiling across Multiple Loop Iterations. SIGPLAN Not. 48, 10 (oct 2013), pp. 373–390. DOI: 10.1145/2544173.2509521.
[7]. Gabriel Marin, Alexey Alexandrov, and Tipp Moseley (2021) Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing. In Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2021). Association for Computing Machinery, New York, NY, USA, pp. 122–133. DOI: 10.1145/3461648. 3463853.
[8]. David Melski and Thomas Reps (1999) Interprocedural Path Profiling. In Compiler Construction, Stefan Jähnichen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 47–62.
[9]. Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019). IEEE Press, pp. 2–14.
[10]. Erez Perelman, Trishul Chilimbi, and Brad Calder. Variational Path Profiling. In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05). IEEE Computer Society, USA, pp. 7–16. DOI: 10.1109/PACT.2005.41
[11]. Subhajit Roy and Y.N. Srikant. Profiling k-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. International Symposium on Code Generation and Optimization (CGO'09), pp. 70–80. DOI: 10.1109/CGO.2009.11
[12]. Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi (2007) Preferential Path Profiling: Compactly Numbering Interesting Paths. In Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07). Association for Computing Machinery, New York, NY, USA, pp. 351–362. DOI: 10.1145/1190216.1190268
[13]. Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani (2004) Structural Path Profiling: An Efficient Online Path Profiling Framework for JustIn-Time Compilers. J. Instruction-Level Parallelism 6.

[14]. Cliff Young and Michael D. Smith. Improving the Accuracy of Static Branch Prediction Using Branch Correlation. SIGOPS Oper. Syst. Rev. 28, 5 (Nov. 1994), 232–241. DOI: 10.1145/381792.195549

[15]. Нейман-заде М. И. and Королёв С. Д. 2020. Руководство по эффективному программированию на платформе «Эльбрус». АО «МЦСТ».

[16]. А.К. Ким, В.И. Перекатов, and С.Г. Ермаков. 2013. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». Питер, СПб.

Информация об авторах / Information about authors

Виктор Евгеньевич ШАМПАРОВ – программист в АО «МЦСТ» с 2017 года, аспирант МФТИ. Сфера научных интересов: оптимизации в компиляторах, профилирование.

Viktor Evgenievich SHAMPAROV – software engineer in MCST Design Center since 2017 and post-graduate student in MIPT. Research interests: compiler optimizations, profiling.

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 of MCST Design Center. His research interests include compiler optimizations, JIT compilation, profiling and accelerated computation libraries.