УДК 004.432

ОПРЕДЕЛЕНИЕ ЗАВИСИМОСТЕЙ ПО ДАННЫМ СРЕДСТВАМИ ДИНАМИЧЕСКОГО АНАЛИЗА СИСТЕМЫ SAPFOR

Н. А. Катаев [1], А. А. Смирнов [2], А. Д. Жуков [3]

[1, 2]*Институт прикладной математики им. М.В. Келдыша РАН, г. Москва;*
[3]*Московский государственный университет им. М.В. Ломоносова, г. Москва*

[1]kataev_nik@mail.ru, [2] smiraland@gmail.com, [3]andreyzkk@yandex.ru

Аннотация

Использование указателей и косвенной адресации в программе, а также сложная структура графа потока управления являются одними из основных препятствий при выполнении статического анализа программ. Обнаруженные в результате такого анализа свойства программы слишком консервативно описывают ее поведение и часто оказываются недостаточными для принятия решений о возможности ее параллельного выполнения. Использование динамического анализа программ позволяет расширить возможности средств автоматизации распараллеливания. В системе SAPFOR (System FOR Automated Parallelization) реализован инструмент динамического анализа, опирающийся на инструментацию программ в представлении LLVM, что позволяет исследовать программы на языках C и Fortran. Чтобы снизить накладные расходы на время выполнения инструментированной программы, сохранив при этом полноту проводимого анализа, используются возможности статического анализа, реализованного в SAPFOR. В процессе динамического анализа часть обращений к памяти, информация о которых была получена в процессе статического анализа, может быть проигнорирована. Разработанный инструмент был протестирован на тестах производительности из пакета NAS Parallel Benchmarks для языков C и Fortran. В процессе динамического анализа кроме традиционных видов зависимостей (flow, anit, output) также определяются переменные, зависимость по которым может быть устранена за счет приватизации или конвейерного выполнения циклов. Совместно с возможностями DVM и OpenMP это существенно облегчает, в том числе, и ручное распараллеливание, облегчая задание соответствующих директив компилятора.

***Ключевые слова**: анализ программ, динамический анализ, автоматизация распараллеливания, SAPFOR, DVM, LLVM*

ВВЕДЕНИЕ

Целью разработки системы автоматизированного распараллеливания SAPFOR [1, 2] (System FOR Automated Parallelization) является снижение сложности разработки параллельных программ. Разностороннее развитие архитектурных и программных составляющих современных вычислительных систем приводит к необходимости совместного использования различных технологий параллельного программирования (MPI, SHMEM, OpenMP, CUDA, OpenACC, OpenCL). Система SAPFOR использует в качестве целевого языка программирования DVM-языки (Fortran-DVMH и C-DVMH), входящие в состав DVM-системы [3, 4]. Данные языки инкапсулируют специфичные для различных программных систем особенности, облегчая как ручную, так и автоматизированную разработку параллельных программ. Разработка системы SAPFOR связана с проведением исследований в трех основных направлениях: анализ программ, автоматическое распараллеливание «хорошо» написанных потенциально параллельных программ и приведение последовательных программ к потенциально параллельному виду. Каждое из этих направлений, как по отдельности, так и в целом, призвано оказать помощь в разработке параллельных программ. Все три упомянутых направления исследований опираются на исследования свойств распараллеливаемой программы. Таким образом, анализ программ является неотъемлемой составляющей любого процесса распараллеливания, не обязательно автоматического.

SAPFOR обеспечивает как статический, так и динамический анализ программ. По отдельности каждого вида анализа оказывается недостаточно. С одной стороны, статический анализ во многих случаях оказывается слишком консервативным, указывая на наличие реально отсутствующих зависимостей с целью сохранения корректности программы. Авторы работы [5], исследуя возможности автоматического распараллеливания в компиляторах Intel и PGI на примере набора модельных приложений для научных расчетов OmpSRC [6], выделяют как минимум две основные проблемы, препятствующие статическому анализу: использование указателей и сложный граф потока управления. В обоих случаях компиляторы вынуждены принимать консервативные решения о наличии зави-

симостей в программе. С другой стороны, динамический анализ, во-первых, анализирует программу для определенного набора входных данных, во-вторых, является ресурсоемким с точки зрения потребляемых времени и памяти.

Кроме факта наличия или отсутствия зависимости в программе, важно понимать возможные способы ее устранения. Например, обратные зависимости и зависимости по выходу могут быть устранены за счет приватизации данных (т. е. создания локальной копии данных для каждого процесса/нити). Соответствующий статический анализ скалярных переменных, основанный на анализе потока данных, реализован в SAPFOR [7]. Но для приватизации массивов необходимо определять множества элементов, используемых на каждой итерации цикла. Адресная арифметика, косвенная индексация, зависимость индексных выражений от параметров функций и переменных, вычисляемых в процессе выполнения программы, приводят к невозможности статического определения таких массивов. При этом устранение такого рода зависимостей является одним из ключевых преобразований, требуемых для распараллеливания тестов производительности из набора NAS Parallel Benchmarks [8, 9]. Другим источником параллелизма являются прямые зависимости по данным, расстояние которых ограничено константой. Циклы с такими зависимостями допускают конвейерное выполнение. Соответствующие директивы ACCROSS предусмотрены в DVM-системе, и их задание требует указания предварительно вычисленного расстояния зависимости. Чтобы справиться с указанными трудностями, в системе SAPFOR был реализован динамический анализ зависимостей по данным.

1. СУЩЕСТВУЮЩИЕ ПОДХОДЫ К ДИНАМИЧЕСКОМУ АНАЛИЗУ

Основными недостатками динамического анализа являются зависимость от полноты входных данных и большие накладные расходы. Степень представительности входных данных существенно влияет на достоверность динамического анализа, использование плохо подобранных наборов данных может приводить к пропуску существующих зависимостей. В этом смысле динамический анализ возможен только под пристальным контролем пользователя, занимающегося распараллеливанием программы, и удобен для использования в системах автоматизации распараллеливания, таких, как SAPFOR.

Для получения информации о программе в процессе динамического анализа широко применяются два следующих подхода: семплирование, то есть получение профиля выполнения программы через заранее заданные промежутки времени, и использование инструментации (вставки в код программы вызовов некоторой внешней библиотеки) для исследования поведения программы в заранее заданных точках. Семплирование значительно менее требовательно к потребляемым ресурсам, но страдает от вероятности потери информации, поэтому для анализа зависимостей по данным оно малоприменимо.

Для определения зависимостей по данным наиболее подходящим подходом является использование инструментации. В свою очередь, вставлять вызовы функций динамического анализа можно на уровне исходного кода, на уровне некоторого внутреннего представления или, выполняя бинарную инструментацию. Инструментация программ на уровне исходного кода требует отдельной реализации инструментов для каждого поддерживаемого языка программирования (в случае SAPFOR это языки Fortran и C). Приходится отдельно обрабатывать все возможные синтаксические конструкции поддерживаемых языков. Кроме того, с целью сокращения накладных расходов в определенных случаях программа может быть предварительно модифицирована и инструментирована не полностью. Такого рода преобразования также удобнее выполнять над некоторым единым для разных языков представлением программы (например, LLVM IR). Бинарная инструментация наиболее эффективна с точки зрения полноты покрытия исполняемой программы, так как допускает инструментацию даже в случае отсутствия исходных кодов, например, позволяя анализировать уже скомпилированные модули, для которых применение других видов инструментации невозможно. Недостатком является то, что возможность соотнесения полученной информации с исходным кодом анализируемой программы сильно зависит от полноты доступной отладочной информации.

В системе SAPFOR в качестве внутреннего представления для анализа программ (как статического, так и динамического) используется LLVM [10]. LLVM IR является универсальным для различных языков программирования, поддерживает возможность выполнения дополнительных анализов и преобразований с целью выборочной инструментации программы, многие из которых уже реализованы в LLVM. Отладочная информация, доступная в LLVM IR, может быть дополнительно

расширена с целью более полного описания исходной программы. С этой целью в SAPFOR обеспечивается соответствие между определенными конструкциями исходной программы, представленными в виде синтаксического дерева (AST) и конструкциями LLVM IR (циклы, переменные, функции и их вызовы). LLVM IR одновременно существует в трех видах: структура классов языка C++ 11, бинарное и текстовое представления. Доступность понятного текстового представления существенно повышает удобство отладки проводимых преобразований и является еще одним преимуществом перед использованием бинарной инструментации. Недостатком является то, что заранее скомпилированные участки программы не могут быть инструментированы и должны подвергаться консервативному анализу.

Одним из наиболее широко применяемых инструментов динамического анализа зависимостей по данным является Intel Advisor, входящий в состав Intel Parallel Studio [11]. Данный инструмент позволяет определять три типа зависимостей по данным: прямые (flow, RAW), обратные (anti, WAR), по выходу (output, WAW). Более детальной классификации, например, с целью выделения переменных которые могут быть объявлены приватными, а также определения расстояний зависимостей, не выполняется. Для анализа используется бинарная инструментация, поэтому с целью корректного соотнесения полученных результатов с исходным кодом рекомендуется отключение оптимизаций. Чтобы запустить анализ зависимостей по данным, нужно предварительно получить профиль выполнения программы: в нем можно будет выбрать циклы, которые должны быть проанализированы. Для получения профиля выполняется семплирование, поэтому количество обнаруженных циклов зависит от размера шага, который используется для сбора данных. Но даже для минимального шага не все циклы будут идентифицированы, например, для программы LU (класс S) при минимальном шаге будет обнаружена только треть циклов (порядка 60 из 187 циклов программы). Это связано в первую очередь с тем, что данные класса S – это тестовые данные очень маленького размера, на которых программа выполняется доли секунды. При этом указанный набор данных достаточно полно описывает поведение программы в целом.

В Таблице 1 приведены замедление Intel Advisor для программы LU (класс S), а также количество циклов, доступных для анализа при минимальном шаге семплирования. Замедление указывается в виде отношения времен выполнения для

программ собранных с разными оптимизационными опциями (-O0 и -O3) с включенным и отключенным режимом анализа зависимостей по данным. Стоит отметить, что рекомендованные режимы анализа предполагают использование опции -O0, чтобы обеспечить анализ исходной программы без оптимизаций и обеспечить точное соответствие выдаваемой информации объектам исходного кода. В этом случае замедление составляет порядка 4160 раз при том, что будет проанализирована только треть всех циклов программы.

Тестирование выполнялось на Intel Xeon CPU E5-1660 v2, 3.70 GHz, с отключенным Turbo Boost. Для компиляции и анализа программ использовалась Intel Parallel Studio 2019 с опорой на системные библиотеки GCC 7.4.

Таблица 1. Замедление Intel Advisor 2019 при анализе теста LU (класс S)

Опции	**-O0 (дин. анализ)**	**-O3 (дин. анализ)**
-O0 (без анализа)	850 раз / 62 цикла	433 раз / 27 циклов
-O3 (без анализа)	**4160 раз / 62 цикла**	2384 раз / 27 цикл

Идея реализованного в SAPFOR алгоритма динамического анализа похожа на алгоритм попарного сравнения (pairwise method) обращений к памяти, описанный в работе [12]. Авторами были предложены улучшения данного алгоритма, позволяющие сократить накладные расходы, но нам не удалось получить доступ к разработанному ими инструменту. При его реализации авторы опираются на бинарную инструментацию, хотя и говорят о возможности использования LLVM. Кроме того, динамический анализ выполнялся не над всеми циклами программы, а только над наиболее ресурсоемкими. Поэтому оценить реальные издержки при анализе всей программы довольно сложно (общее количество циклов в программах не приводится). Авторы статьи выполняли анализ с целью дальнейшего распараллеливания для систем с общей памятью, и в этом случае выборочный анализ циклов оказывается оправданным. В SAPFOR необходимо принимать глобальные решения о распределении данных и вычислений, следовательно, нельзя исключать циклы из анализа, так как они могут обращаться к распределяемым данным (в том числе, косвенно, то есть средствами статического анализа отследить данные ситуации может быть невозможно).

2. АЛГОРИТМ ДИНАМИЧЕСКОГО АНАЛИЗА

В рамках системы SAPFOR требуется, чтобы динамический анализ обнаруживал для каждого цикла программы тип зависимости по данным и возможность ее устранения. Для определения возможности конвейерного выполнения цикла требуется знать расстояние зависимости (максимальное и минимальное). Также необходимо определять ситуации, когда зависимость может быть устранена за счет приватизации данных. В этом случае кроме локального анализа тела цикла необходимо убедиться, что значение переменной, вычисленное в цикле, не используется после выхода из него.

Так как не требуется определять конкретные операторы, порождающие зависимости по данным, можно не хранить список всех обращений к памяти, а только сам факт обращения к данной ячейке памяти и некоторую дополнительную информацию, необходимую для выявления желаемых свойств. При этом в момент обращения к ячейке памяти приходится выполнять элементарную обработку накопленных данных об этой ячейке. Такой подход позволяет снизить требования к памяти в случае, когда на каждой итерации цикла происходят множественные обращения по одному и тому же адресу. Время динамического анализа изменяется неочевидным образом, поскольку с одной стороны в момент обращения происходит небольшая обработка текущих данных, что немного замедляет выполнение, с другой стороны, после выхода из цикла требуется обработать меньший объем накопленных данных, что ускоряет работу. Поэтому время работы сильно зависит от структуры анализируемой программы.

Для каждого вложенного цикла имеется своя собственная информация о памяти, с которой была зафиксирована работа в рамках этого цикла. В момент обращения к памяти информация о ней обновляется только в самом вложенном цикле. После выхода из цикла накопленная информация используется для обновления данных в объемлющем цикле, а также запоминаются найденные зависимости для данного цикла в глобальном хранилище.

Динамический анализатор использует две основные структуры данных: глобальное хранилище результатов анализа и стек контекстов. В глобальном хранилище результатов анализа для каждого цикла программы содержится информация обо всех найденных зависимостях: тип зависимости (прямая, обратная, по выходу),

расстояние зависимости (максимальное и минимальное), возможность приватизации переменной, вызвавшей зависимость. Контекстом называется множество адресов памяти в связке с дополнительной информацией, необходимой для выявления интересующих свойств. При входе в очередной цикл или функцию анализируемой программы создается новый пустой контекст и добавляется в стек. В контексте хранится текущая итерация соответствующего цикла. В случае, когда контекст соответствует функции, итерация не имеет значения и может быть любой. При переходе на следующую итерацию цикла вызывается функция библиотеки анализа, которая изменяет итерацию в соответствующем контексте. В момент обращения к памяти в верхнем контексте стека находится или создается новый объект с информацией об обращениях по данному адресу, при этом используется текущая итерация, хранящаяся в контексте. Например, если по адресу происходит чтение, то в контексте находится объект, соответствующий указанному адресу, и в нем отмечается, что последнее чтение было произведено на такой-то итерации. Если при этом записано, что на другой итерации имелась запись, то фиксируется факт зависимости и вычисляется расстояние между итерациями. Для вычисления расстояния необходимо хранить не только номер итерации последнего чтения/записи, но и номер итерации первого чтения/записи.

При выходе из цикла или функции происходит удаление контекста из стека. Для каждого адреса из удаленного контекста, если было зафиксировано чтение/запись по этому адресу, фиксируются соответствующие операции в вершине стека. В глобальное хранилище добавляется информация об обнаруженных зависимостях для цикла, соответствующего удаленному контексту.

Определение факта использования переменной после цикла происходит с использованием других структур данных, за исключением глобального хранилища.

Во время выполнения программы при выходе из цикла выполняются следующие действия. Для каждой переменной, к которой производилось обращение в цикле (в том числе, и неявно, внутри вызываемых функций), запоминается адрес структуры данных в глобальном хранилище, содержащей информацию о зависимостях в данном цикле. Запомненные адреса хранятся в списке, который назовем «списком циклов». Для каждой переменной будет создан свой список циклов.

При обращении к переменной происходит поиск соответствующего ей списка циклов. В случае отсутствия списка не требуется предпринимать каких-либо действий. Если же список найден и к переменной обратились на чтение, то для каждого цикла в списке отмечается, что данная переменная используется после этого цикла. После указанных действий список удаляется.

Для корректной обработки автоматических переменных, располагающихся на стеке, необходимо при выходе из функции удалять списки циклов, соответствующие этим локальным переменным.

В момент завершения программы информация из глобального хранилища выдается в заданном формате.

3. ДЕТАЛИ РЕАЛИЗАЦИИ

Библиотека динамического анализа реализована на C++11, инструментация выполняется на уровне LLVM IR. Инструментация заключается во вставке вызовов функций библиотеки динамического анализа. Для каждого инструментируемого объекта динамическому анализатору передается описывающая его метаинформация. Данная информация может быть использована в динамическом анализаторе для определения объекта исходного кода, соответствующего инструментированному объекту. Поддерживается инструментация для программ на языках Fortran и C/C++. LLVM IR не зависит от исходного языка, но отладочная информация, используемая для описания инструментируемых объектов, может несколько отличаться. Например, для описания COMMON-блоков языка Fortran используются структуры специального вида. Поэтому для других языков может отсутствовать высокоуровневое описание инструментируемых данных.

Динамический анализ состоит из следующей последовательности шагов:

1. Получение LLVM IR для каждого файла, который должен быть проанализирован;

2. Компоновка полученных файлов, содержащих LLVM IR, с помощью инструмента llvm-link, входящего в состав LLVM;

3. Инструментация полученного единого файла (модуля LLVM IR);

4. Компиляция инструментированного файла, компоновка его с остальными частями анализируемого проекта, а также с библиотекой динамического анализа.

Результатом запуска полученного исполняемого файла будет либо текстовое описание всех найденных зависимостей по данным, либо описание зависимостей в формате JSON. Желаемый способ представления результатов анализа может быть задан с помощью переменных окружения непосредственно перед исполнением анализируемой программы.

Структура генерируемого JSON-файла приведена в Таблице 2 и включает два основных блока информации: список переменных, для которых выполнялся анализ, и информация о результатах анализа для каждого инструментированного цикла программы. Полученный JSON-файл может быть передан статическому анализатору системы SAPFOR для уточнения результатов анализа. Для соотнесения результатов динамического анализа с внутренним представлением программы в системе SAPFOR используется информация о расположении объектов в исходном коде программы (имя файла, номер строки и столбца), для переменных дополнительно используются их имена.

В разделе приватизируемых переменных указываются, во-первых, переменные, использование которых локализовано в рамках каждой итерации цикла (т. е. для каждой итерации цикла может быть создана своя копия переменной), во-вторых, факт использования значений переменных, полученных на какой-либо итерации цикла после выхода из цикла (UseAfterLoop). Спецификации типа *private*, доступные как в DVMH-языках, так и в OpenMP, приводят к созданию локальных копий исходных переменных, таким образом, при параллельном выполнении после выхода из цикла исходные переменные будут иметь значения, отличные о тех, которые они имели бы при последовательном выполнении. Для указания таких ситуаций используется свойство UseAfterLoop, в этом случае распараллеливание может потребовать дополнительных действий, и применение только спецификации *private* не допустимо. Например, если известно, что используемое после выхода из цикла значение переменной было вычислено на последней итерации цикла, то можно использовать спецификацию *lastprivate* OpenMP-языков.

Таблица 2. Структура JSON-файла отражающего результаты динамического анализа программы

{ "Vars": [{ "File": <path-to-file>, "Line": <number>, "Column": <number>, "Name": <name>, }, "Loops": [{ "File": <path-to-file>, "Line": <number>, "Column": <number>,	Список зарегистрированных переменных. Список зарегистрированных циклов.
"Write": [<var-list>], "Read": [<var-list>],	Режим доступа к зарегистрированным переменным
"Private": [<var-list>], "UseAfterLoop": [<var-list>],	Приватизируемые переменные.
"Flow": [{<var-id> : {"Min": <distance>, "Max": <distance>},...], "Anti": [{<var-id> : {"Min": <distance>, "Max": <distance>},...], "Output": [{<var-id> : {"Min": <distance>, "Max": <distance>},...] } }	Зависимости по данным с указанием расстояние зависимости.

Дополнительным источником параллелизма в циклах является наличие прямых и обратных зависимостей по данным, расстояние которых ограничено константой. Для параллельного выполнения таких циклов DVMH-языки предусматривают спецификацию *across*, а OpenMP-языки – спецификацию *ordered*. Для использования обеих спецификаций требуется явно указать, между какими итерациями цикла существует зависимость. Необходимая для этого информация будет указана при описании зависимостей по данным в результатах динамического анализа. В приведенном в Таблице 3 примере присутствуют как прямая, так и обратная зависимости по данным расстояния один, которое в явном виде указано в спецификациях *across* и *ordered*.

Таблица 3. Пример использования спецификаций across и ordered для параллельного выполнения гнезда циклов с зависимостями

```
#pragma dvm parallel([i][j] on A[i][j]) across(A[1:1][1:1])
for (i = 1; i < N-1; i++)
  for (j = 1; j < N-1; j++)
    A[i][j]=(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
```

```
#pragma omp parallel for ordered(2)
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++) {
    #pragma omp ordered depend (sink: i - 1, j) depend (sink: i, j - 1)
    A[i][j]=(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
    #pragma omp ordered depend (source)
}
```

Инструментация модуля LLVM IR включает в себя:

- вставку объявлений функций динамического анализатора;
- вставку вызовов данных функций в тела функций инструментируемого модуля;
- объявление глобального пула, содержащего мета информацию, описывающую инструментируемые объекты модуля;
- создание вспомогательных функций, отвечающих за инициализацию мета информации, регистрацию типов и глобальных данных;
- вставку вызовов функций инициализации в начало функции, являющейся точкой входа программы.

В процессе инструментации регистрируются вызовы функций, обращения к памяти на чтение и запись, начало, конец, а также начало каждой итерации циклов, связь между фактическими и формальными параметрами.

4. ПРИМЕНЕНИЕ ДИНАМИЧЕСКОГО АНАЛИЗА

Разработанный инструмент был протестирован на тестах производительности из пакета NAS Parallel Benchmarks в версиях 3.3.1 для языков Fortran [8] и C [9]. Оценка накладных расходов на время выполнения приведена на Рис. 2 и Рис. 3. Рост потребляемой памяти (в количестве раз) указан на Рис. 4 и 5.

Тестирование выполнялось на Intel Xeon CPU E5-1660 v2, 3.70 GHz, с отключенным Turbo Boost. Для получения LLVM IR и компиляции программ использовались компиляторы Clang и Flang версий 7.1.0 с опорой на библиотеки компилятора GCC 7.4. Компиляция выполнялась с использованием опции –O3. В отличие от применения бинарной инструментации использование данной опции допускается, так как входящие в ее состав оптимизации применяются уже после инструментации исходной программы. Также приведены накладные расходы при использовании Intel Advisor. В данном случае анализ зависимостей выполнялся с опцией –O0, чтобы гарантировать получение достоверных результатов. При этом замедление указано относительно программ, собранных с опцией –O3.

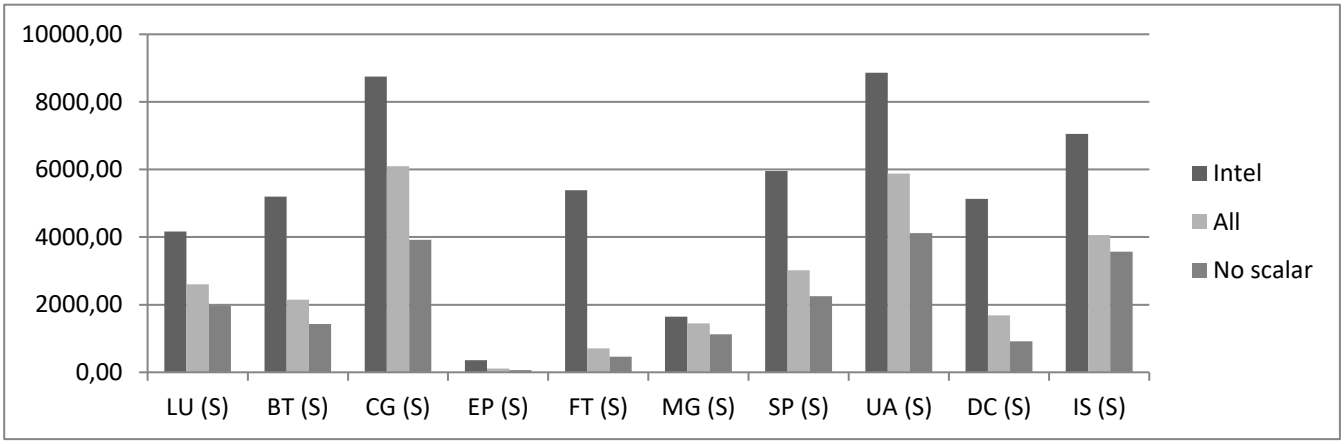


Рис. 2. Замедление C-тестов из набора NAS Parallel Benchmarks при полной (All) и выборочной (No scalar) инструментациях и использовании Intel Advisor

С целью снижения накладных расходов были использованы реализованные в SAPFOR средства статического анализа [7]. В большинстве случаев их оказывается достаточно для анализа скалярных переменных, к которым не применяются операции взятия адреса и которые не участвуют в операциях адресной арифметики.

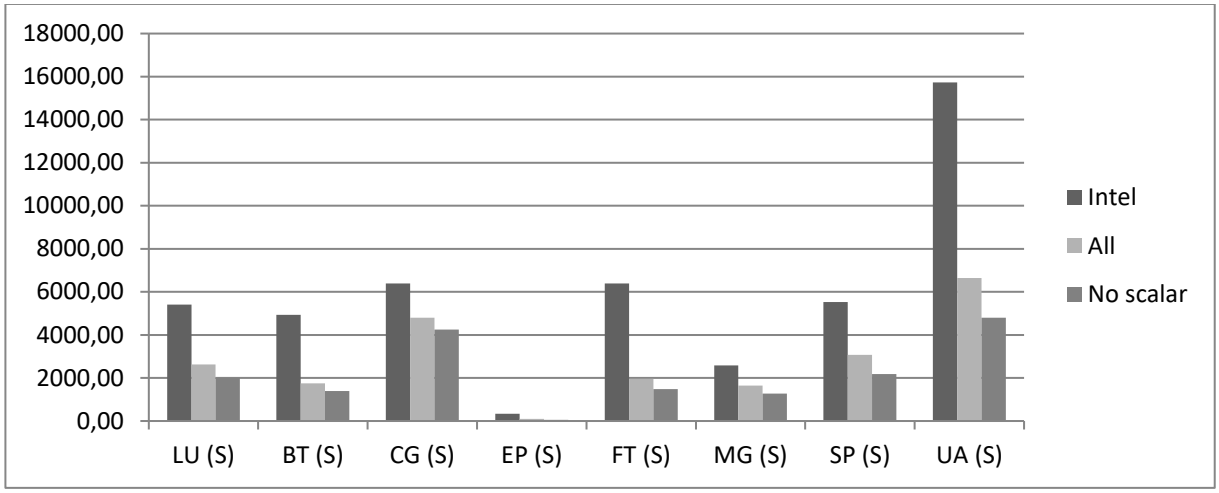


Рис. 3. Замедление Fortran-тестов из набора NAS Parallel Benchmarks при полной (All) и выборочной (No scalar) инструментациях и использовании Intel Advisor

Для зависимостей по данным переменным выполняется их классификация с целью устранения зависимостей за счет использования редукционных операций и приватизации соответствующих данных. Данные переменные могут быть проигнорированы в процессе динамического анализа, более того, обращения к данным переменным могут быть оптимизированы средствами LLVM, например, они могут быть размещены на регистрах. Применение данной оптимизации позволило сократить замедление анализируемых программ до 40% (на тесте EP).

Таким образом, с учетом данной оптимизации время выполнения увеличивается в среднем до 2000 раз. Но совместно с применением статического анализа обеспечивается полный анализ всей программы. При этом анализ трети всех циклов, выполняемый средствами Intel Advisor, приводит к замедлению программы в среднем до 5000 раз.

Также была реализована дополнительная возможность, позволяющая выбрать функцию, начиная с которой должна выполняться инструментация. В данном случае будут проанализированы указанная функция, а также все функции, которые из нее вызываются. Выборочный анализ отдельных функций, важных для распараллеливания конкретного цикла, значительно ускоряет анализ программ. Например, для теста LU (класс S) рост времени для анализа одного из основных циклов с регулярной зависимостью по данным, которую можно устранить за счет конвейерного выполнения цикла, составляет 707 раз.

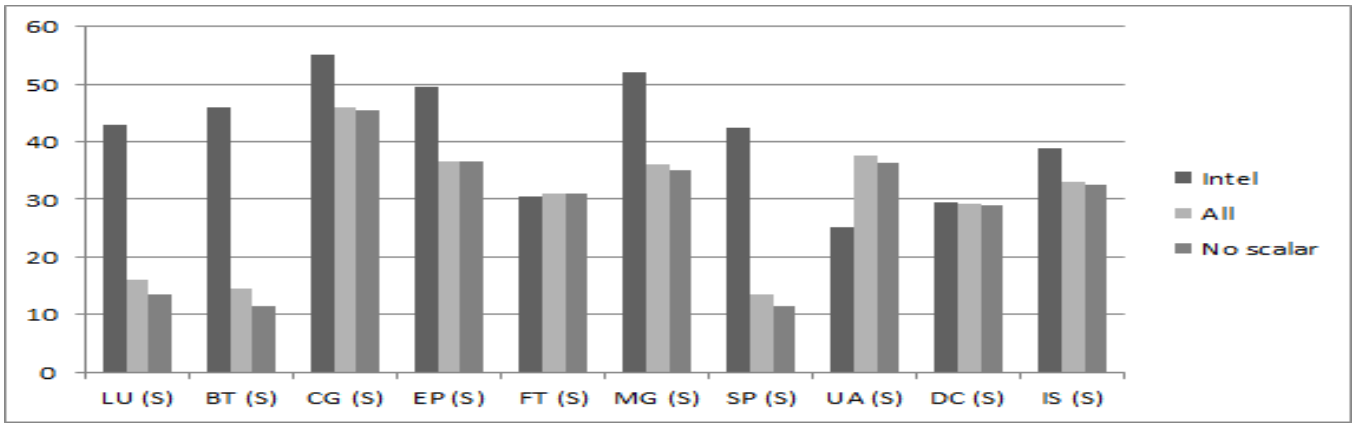


Рис. 4. Рост потребления памяти C-тестов из набора NAS Parallel Benchmarks при полной (All) и выборочной (No scalar) инструментациях и использовании Intel Advisor

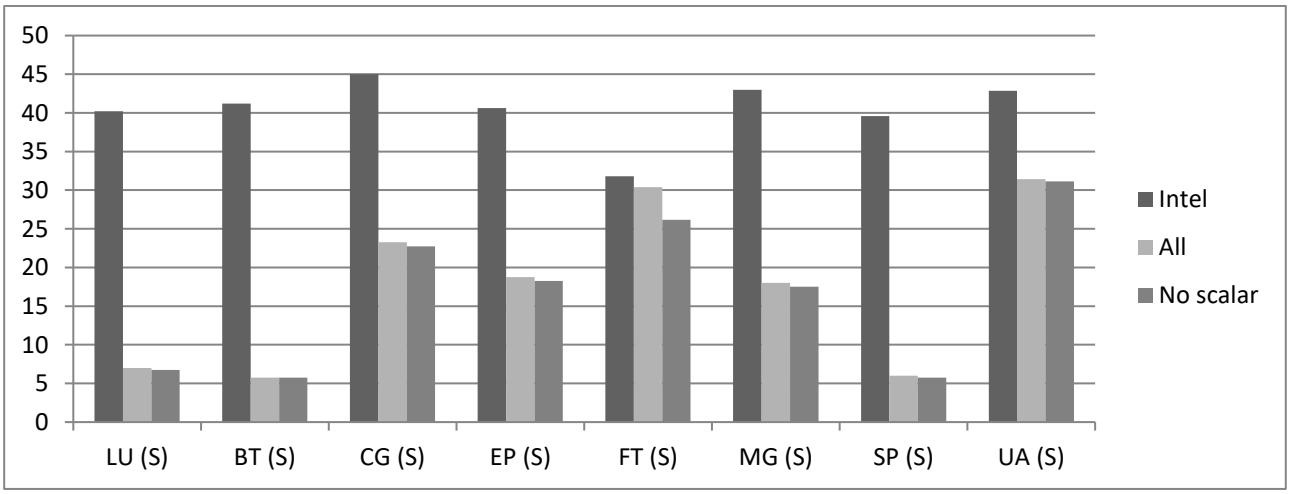


Рис. 5. Рост потребления памяти Fortran-тестов из набора NAS Parallel Benchmarks при полной (All) и выборочной (No scalar) инструментациях и использовании Intel Advisor

ЗАКЛЮЧЕНИЕ

В работе рассмотрен инструмент, предназначенный для динамического анализа зависимостей по данным, реализованный в рамках системы SAPFOR. Инструмент может быть использован как для получения результатов анализа с целью автоматизации распараллеливания программ в системе SAPFOR, так и с целью ручного распараллеливания программ. Помимо определения основных типов зависимостей (flow, anti, output) также предоставлена рекомендация о том, какие зависимости могут быть устранены за счет приватизации переменных, в том числе, массивов

(что особенно важно из-за ограниченности возможностей статического анализа), а также о том, для каких циклов может быть организовано конвейерное выполнение. При этом предоставляемой информации о расстоянии зависимостей достаточно для расстановки директив ACROSS DVM-языков, позволяющих организовать конвейер автоматически. Статический анализ скалярных переменных позволил снизить накладные расходы на проведение динамического анализа, не потеряв полноты результатов. Использование инструментации LLVM IR вместо бинарной инструментации помимо проведения предварительного статического анализа позволяет выполнять оптимизацию программы после инструментации без потери точности результатов и их соотнесения с объектами исходного кода. Дальнейшие исследования планируется направить на большее снижение накладных расходов, так как распараллеливание для вычислительных систем с распределенной памятью требует анализа всей программ, сильно ограничивая возможности выборочного анализа циклов.

Исходные коды системы SAPFOR доступны на GitHub [13].

СПИСОК ЛИТЕРАТУРЫ

1. *Клинов М.С., Крюков В.А.* Автоматическое распараллеливание Фортран-программ. Отображение на кластер // Вестник Нижегородского университета им. Н.И. Лобачевского, 2009. № 2. С. 128–134.
2. *Бахтин В.А., Жукова О.Ф., Катаев Н.А., Колганов А.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н., Савицкая О.А., Смирнов А.А.* Автоматизация распараллеливания программных комплексов // Труды XVIII Всероссийской научной конференции «Научный сервис в сети Интернет», Новороссийск, Россия, 19–24 сентября 2016 г. М.: ИПМ им. М.В. Келдыша, 2016. С. 76–85. doi:10.20948/abrau-2016-31
3. *Konovalov N.A., Krukov V.A, Mikhajlov S.N., Pogrebtsov A.A.* Fortan DVM: a Language for Portable Parallel Program Development // Programming and Computer Software. 1995. V. 21. No. 1. P. 35–38.
4. *Бахтин В.А., Клинов М.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н., Сазанов Ю.Л.* Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами // Вестник Южно-Уральского государственного университета, серия «Математическое моделирование и про-

граммирование», 2012. №18 (277), выпуск 12. Челябинск: Издательский центр ЮУрГУ. С. 82–92.

5. *Kim M., Kim H., Luk C.K.* Prospector: A dynamic data-dependence profiler to help parallel programming // HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism, 2010.
6. *Dorta A.J., Rodríguez C., de Sande F., Gonzalez-Escribano A.* The OpenMP Source Code Repository // Parallel, Distributed, and Network-Based Processing, Euromicro Conference, 2005.
7. *Kataev N.A.* Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR // Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, 2018. Vol 965. Springer, Cham. P. 487-499. doi:10.1007/978-3-030-05807-4_41
8. NAS Parallel Benchmarks. UFL: https://www.nas.nasa.gov/publications/npb.html
9. *Seo S., Jo G., Lee J.* Performance Characterization of the NAS Parallel Benchmarks in OpenCL // 2011 IEEE International Symposium on. Workload Characterization (IISWC), 2011. P. 137–148.
10. *Lattner C., Adve V.* LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, 2004.
11. Intel Parallel Studio. URL: https://software.intel.com/en-us/parallel-studio-xe
12. *Kim M., Kim H., Luk C.K.* SD3: A Scalable Approach to Dynamic Data-Dependence Profiling // 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2011. doi:10.1109/MICRO.2010.49
13. SAPFOR. URL: https://github.com/dvm-system.

INVESTIGATION OF DATA DEPENDENCIES BY DYNAMIC ANALYSIS OF SAPFOR

N.A. Kataev [1], A.A. Smirnov [2], A.D. Zhukov [3]

[1,2]*Keldysh Institute of Applied Mathematic RAS;* [3] *Lomonosov Moscow State University*

[1]kataev_nik@mail.ru, [2] smiraland@gmail.com, [3]andreyzkk@yandex.ru

Abstract

The use of pointers and indirect memory accesses in the program, as well as the complex control flow are some of the main weaknesses of the static analysis of programs. The program properties investigated by this analysis are too conservative to accurately describe program behavior and hence they prevent parallel execution of the program. The application of dynamic analysis allows us to expand the capabilities of semi-automatic parallelization. In the SAPFOR system (System FOR Automated Parallelization), a dynamic analysis tool has been implemented, based on on the instrumentation of the LLVM representation of an analyzed program, which allows the system to explore programs in both C and Fortran programming languages. The capabilities of the static analysis implemented in SAPFOR are used to reduce the overhead program execution, while maintaining the completeness of the analysis. The use of static analysis allows to reduce the number of analyzed memory accesses and to ignore scalar variables, which can be explored in a static way. The developed tool was tested on performance tests from the NAS Parallel Benchmarks package for C and Fortran languages. The implementation of dynamic analysis, in addition to traditional types of data dependencies (flow, anit, output), allows us to determine privitizable variables and a possibility of pipeline execution of loops. Together with the capabilities of DVM and OpenMP these greatly facilitates program parallelization and simplify insertion of the appropriate compiler directives.

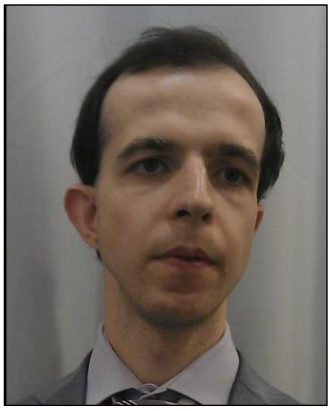
Keywords*: program analysis, dynamic analysis, semi-automatic parallelization, SAPFOR, DVM, LLVM*

REFERENCES

1. *Klinov M.S., Kriukov V.A.* Avtomaticheskoe rasparallelivanie Fortran-programm. Otobrazhenie na klaster // Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo, 2009. No 2. S. 128–134.
2. *Bakhtin V.A., Zhukova O.F., Kataev N.A., Kolganov A.S., Kriukov V.A., Podderiugina N.V., Pritula M.N., Savitskaia O.A., Smirnov A.A.* Avtomatizatsiia rasparallelivaniia programmnykh kompleksov // Trudy XVIII Vserossiiskoi nauchnoi konferentsii «Nauchnyi servis v seti Internet», Novorossiisk, Russia, 19–24 sentiabria. M.: IPM im. M.V. Keldysha, 2016. P. 76–85. doi:10.20948/abrau-2016-31
3. *Konovalov N.A., Krukov V.A, Mikhajlov S.N., Pogrebtsov A.A.* Fortan DVM: a Language for Portable Parallel Program Development // Programming and Computer Software, 1995. V. 21. No. 1. P. 35–38.
4. *Bakhtin V.A., Klinov M.S., Kriukov V.A., Podderiugina N.V., Pritula M.N., Sazanov Iu.L.* Rasshirenie DVM-modeli parallelnogo programmirovaniia dlia klasterov s geterogennymi uzlami // Vestnik Iuzhno-Uralskogo gosudarstvennogo universiteta, seriia "Matematicheskoe modelirovanie i programmirovanie", No 18 (277), vypusk 12. Cheliabinsk: Izdatelskii tsentr IuUrGU, 2012. S. 82–92.
5. *Kim M., Kim H., Luk C.K.* Prospector: A dynamic data-dependence profiler to help parallel programming // HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism, 2010.
6. *Dorta A.J., Rodríguez C., de Sande F., Gonzalez-Escribano A.* The OpenMP Source Code Repository // Parallel, Distributed, and Network-Based Processing, Euromicro Conference, 2005.
7. *Kataev N.A.* Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR // Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, 2018. Vol 965. Springer, Cham. P. 487–499. doi:10.1007/978-3-030-05807-4_41
8. NAS Parallel Benchmarks. URL: https://www.nas.nasa.gov/publications/npb.html
9. *Seo S., Jo G., Lee J.* Performance Characterization of the NAS Parallel Benchmarks in OpenCL // 2011 IEEE International Symposium on. Workload Characterization (IISWC), 2011. P. 137–148.

10. *Lattner C., Adve V.* LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, 2004.
11. Intel Parallel Studio. URL: https://software.intel.com/en-us/parallel-studio-xe
12. *Kim M., Kim H., Luk C.K.* SD3: A Scalable Approach to Dynamic Data-Dependence Profiling // 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2011. doi:10.1109/MICRO.2010.49
13. SAPFOR. URL: https://github.com/dvm-system

СВЕДЕНИЯ ОБ АВТОРАХ



КАТАЕВ Никита Андреевич – научный сотрудник ИПМ им. М.В. Келдыша, специалист в области системного программирования. Сфера научных интересов – компиляторные технологии, автоматизация распараллеливания программ.

Nikita Andreevich KATAEV – Researcher of KIAM RAS, a specialist in system programming. Research interests include compiler technologies, semi-automatic program parallelization.

email: kataev_nik@mail.ru

СМИРНОВ Александр Андреевич – научный сотрудник ИПМ им. М.В. Келдыша, специалист в области системного программирования. Сфера научных интересов – компиляторные технологии, автоматизация распараллеливания программ.

Alexander Andreevich SMIRNOV – Researcher of KIAM RAS, a specialist in system programming. Research interests include compiler technologies, semi-automatic program parallelization.

email: smiraland@gmail.com

ЖУКОВ Андрей Дмитриевич – студент 2 курса магистратуры факультета ВМиК МГУ им М.В. Ломоносова.

Andrey Dmitrievich ZHUKOV – student of CMC faculty of Lomonosov Moscow State University

email: andreyzkk@yandex.ru

Материал поступил в редакцию 15 ноября 2019 года