Дроздов А.Ю., Особенности и перспективы Универсальной технологии оптимизирующей компиляции. // Сборник научных трудов ИТМиВТ им. С.А. Лебедева РАН, Выпуск №1, Материалы Всероссийской конференции «Перспективы развития высокопроизводительных архитектур. История, современность и будущее отечественного компьютеростроения», 2008, С. 62-74.

Введение

Современное состояние ИТ индустрии характеризуется огромным разнообразием вычислительных систем. Основным инструментом, позволяющим эффективно использовать аппаратные возможности вычислительных систем, является оптимизирующий компилятор ([1-4, 6). Основная задача оптимизирующего компилятора – получение кода, максимально эффективного для данного вычислительного комплекса.

Разработка каждого отдельного решения требует временных затрат в пределах десятков лет и значительных затрат по сопровождению этих решений. (примеры: оптимизирующие компиляторы компаний Intel, Sun Microsystems, Transmeta, Microsoft, IBM, HP, Elbrus, …). Все эти компании создали свои собственные, не совместимые друг с

другом коммерческие продукты, в которые их разработчики не способны оперативно встраивать поддержку новых аппаратных решений. и требований.

Имеющиеся открытые системы, такие как GNU технология ([9]), хоть и позволяют быстро сделать сквозную технологию компиляции для каждой конкретной архитектуры, тем не менее, не в состоянии обеспечить приемлемый уровень качества (производительности) использования аппаратных ресурсов для новейших архитектур процессоров, таких как многоядерные процессоры и процессоры с явным параллелизмом на уровне команд. Это подталкивает коммерческие компании к разработке собственных оптимизирующих компиляторов, которые решают проблемы эффективности для отдельно взятой архитектуры.

В ИТМиВТ была поставлена задача создания универсальной технологии оптимизирующей компиляции, которую можно портировать в состав любой существующей системы компиляции. Для достижения поставленной цели были разработаны технологические и алгоритмические основы создания оптимизирующих компиляторов, которые можно использовать в качестве набора инструментов для быстрого и эффективного построения оптимизирующих компиляторов любых типов и конфигураций. Разработанные технологии и алгоритмы (блоки оптимизирущей компиляции) позволют:

- улучшать существующие оптимизирующие компиляторы
- создавать автоматические распараллеливатели для многоядерных архитектур
- создавать новые оптимизирующие компиляторы
- создавать оптимизирующие компиляторы времени исполнения
- оценивать эффективность новых архитектур
- облегчать освоение новых технологий программирования
- помогать в разработке новых языков программирования
- помогать в учебном освоении технологий оптимизирующей компиляции

Блоки могут быть использованы везде, где нужно оптимизировать работу приложений, получить максимальную эффективность работы вычислительных ресурсов. Эту задачу удалось успешно решить и основными частями решения поставленной задачи стали:

- разработка технологии построения оптимизирующих компиляторов, основанной на разделении всей функциональности на неэвристическую и эвристическую части и разбиение функциональности на различные уровни абстракции
- разработка технологии реализации блоков оптимизирующей компиляции; позволяющей эффективно распараллеливать работу компилятора
- разработка технологии портирования блоков оптимизирующей компиляции в контекст произвольной существующей инфраструктуры оптимизирующей компиляции и апробация этой технологии на примере автоматического распараллеливателя программ, встроенного в технологическую цепочку GCC
- разработка алгоритмических основ компиляторостроения в таких важнейших и критических областях как статический анализ программ, планирование программ, многопоточное распараллеливание программ
- разработка готовых продуктов на основе разработанных технологических и алгоритмических основ оптимизирующего компиляторостроения

1 Технологические аспекты реализации блоков оптимизирующей компиляции

1.1 Инфраструктурные аспекты

В этом разделе вкратце остановимся на инфраструктурных аспектах реализации технологий оптимизирующей компиляции.

В качестве языка реализации был выбран язык C++, как проверенный инструмент отображения абстракций объектно-ориентированного проектирования.

Документация создается в автоматической системе документирования Doxygen. Вся документация размечена XML тегами, что позволяет скомпилировать документацию для разных целей в разных видах с разным уровнем детализации.

Часть документации компилируется непосредственно из кодов, часть - из описаний. Можно, например, скомпилировать отдельно описания интерфейсов или только высокоуровневую информацию по блокам. Документация всегда поддерживается в свежем состоянии.

Важнейшей составляющей успеха настоящей работы (широкого применения предлагаемой технологии) является высочайшее качество компонент. Для его обеспечения используются последние достижения в области программной инженерии, такие как:

- использование единого стиля программирования
- использование методов и практик объектно-ориентированного проектирования и программирования
- использование средств автоматического обнаружения ошибок путем инструментирования кода программ с последующим исполнением
- использование средств внутренней верификации и отладки
- использование средств визуализации внутренних структур данных
- использование средств автоматической документации программ

В работе используются новейшие методы тестирования:

- тестирование каждого блока отдельно от других (unit testing)
- тестирование на наличие ожидаемой функциональности (regression testing)
- тестирование на покрытие
- тестирование с помощью автоматической генерации тестов
- стрессовое тестирование
- просмотр исходных текстов инженерами, работающими над смежными блоками (code review)
- регулярные собрания по существующим открытым ошибкам

Также необходимо упомянуть о важности правильной организации всего процесса разработки, который должен включать в себя такие этапы как – проектирование, кодирование, отладку, документирование, анализ производительности, контроль деградации производительности и тестирование. Для поддержания высоких стандартов качества работы и качественных результатов, в работу проектных групп необходимо внедрять наиболее передовые и эффективные методики международных стандартов проектного менеджмента и управления жизненным циклом программных продуктов.

Бесперебойная, синхронная работа этого процесса является гарантией своевременного выпуска готовой продукции и своевремменого поступления продукции потребителям.

На настоящий момент времени блоки оптимизирующей компиляции, реализованные на основе технологии, предложенной в данной работе, оттестированы в технологической цепочке компиляторов GCC. Компиляция с языков C, C++, FORTRAN оттестирована в в контексте архитектуры x86. Сейчас есть возможность производить тестирование в контексте любой целевой архитектуры, поддержанной в технологической цепочке GCC - x86, Itanium, SPARC, MIPS, PowerPC, Cell ([9]).

1.2 Параметризация окружения

Рис. 1 иллюстрирует основной подход к переносимости технологии в произвольные технологические цепочки.

Блоки используют информацию об окружении, хранящуюся в Базе Данных программы (БДП). База Данных программы хранит семантическое представление программы, высокоуровневую информацию, символьные таблицы, типы, размерности и т.д. – всё, что существует в языках высокого уровня, а также включает описание целевой архитектуры.

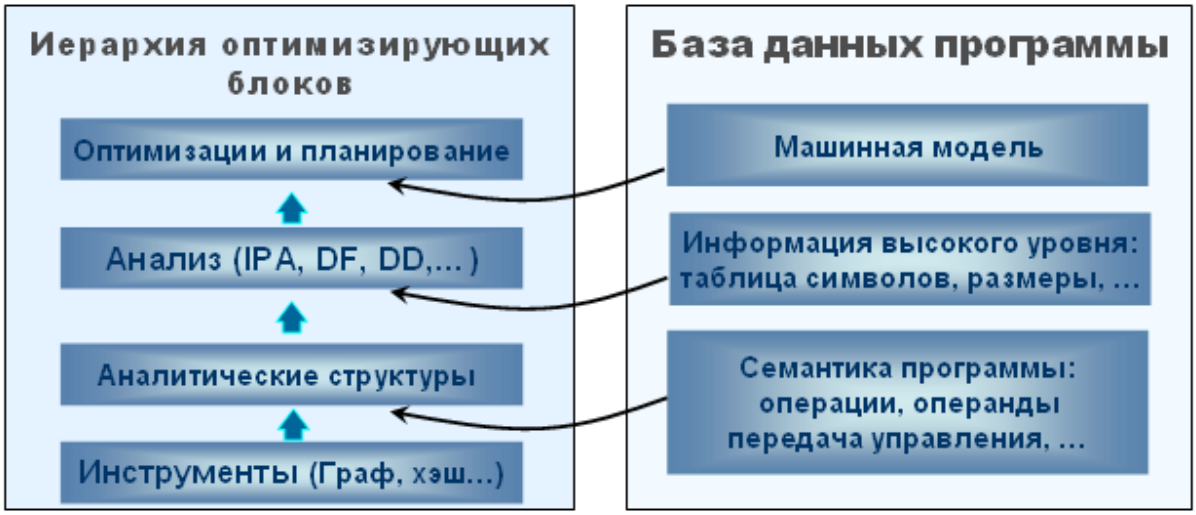

Рис.1 Иерархия блоков и база данных программы

Блоки имеют иерархическую структуру - каждый следующий уровень использует результаты предыдущего уровня. Основу составляют инструментальные блоки, результаты их работы используются блоками моделей, модели используются алгоритмами аланиза, а результаты анализа применяются для оптимизации и планирования, необходимого для данной архитектуры.

Параметризация оптимизирующего компилятора при помощи базы данных позволяет переносить решение полностью в любые технологические цепочки – как новые, так и уже существующие. Каждый функциональный элемент может использоваться отдельно.

Одна из основных целей данной работы в части создания блоков оптимизирующей компиляции состоит в том, что все блоки свободно встраиваемы в любую среду технологическую среду компиляции. Для этого необходимо абстрагировать реализацию блоков от промежуточного представления программы, которое в настоящий момент существует в любой инфраструктуре, все алгоритмы пишутся с использованием конкретных особенностей реализации этого представления и, таким образом, становятся полностью непереносимыми в другую среду.

Все существующие подходы выносят промежуточное представление программы во внешний интерфейс и все алгоритмы реализуются на основе этих интерфейсов, что не позволяет использовать эти алгоритмы где-либо еще. Например, то, что было уже реализовано в контексте gcc, не может быть использовано в других технологических цепочках без полного или частичного переписывания алгоритмов. Предлагаемый подход решает эту проблему полностью. Для этого вся информация об окружении, которая содержится во внутренних представлениях gcc или другой оптимизирующей технологии, сбрасывается в формат базы данных программы. На базе этой информации работают

алгоритмы, результат работы которых снова представляется в терминах базы данных, из которой опять при желании можно вернуться в представление gcc или любое другое.

Для того чтобы произвести абстрагирование, программу необходимо представить в понятиях настолько общих, что внутренняя структура реализации этого представления не будет никак сказываться на реализации блоков. Наиболее общий подход к хранению и манипуляции данных, позволяющий практически полностью абстрагировать конкретный способ хранения этих данных это базы типа Oracle. Доступ к базе данных (заполнение, изменение, удаление) осуществляется при помощи интерфейса запросов (SQL) или при помощи функционального интерфейса (PL SQL) что и позволяет полностью скрыть реализацию базы от пользователя. Подобный подход применим и в случае с информацией о программе (информация о высокоуровневых языках, информация о семантике операций и информация о целевой архитектуре (machine model)). Задание всех входных параметров компиляции через такую базу данных позволяет нам полностью скрыть от пользователя детали используемого нами промежуточного представления программы и, таким образом, делает все блоки независимыми от технологической цепочки, в которой они будут использоваться.

Естественно это не совсем бесплатное свойство. Для того чтобы использовать эти блоки в произвольной технологической цепочке, от пользователя требуется передавать всю необходимую информацию о программе и ее окружении в стандарте разработанной предлагаемой базы данных о программе.

Итак, мы имеем три разновидности информации о контексте, в котором работает оптимизатор: информация с языка высокого уровня (измерения массивов, символьная таблица, и может быть что-то еще), информация о семантике (представление семантики через небольшую номенклатуру операций, с заданием порядка между ними), и информация о машинной модели (информация о правилах, по которым команды должны помещаться в исполняемый файл, а также вся информация об особенностях архитектуры, которую можно использовать в целях оптимизации). Таким образом, мы имеем полную гибкость в отношении контекста. Например, в случае двоичной компиляции в базе не будет ссылок из семантической в таблицу языковой информации, но будут ссылки в таблицу машинной модели и все методы оптимизации будут работать без каких-либо изменений. Это также относится к случаю, когда есть только семантика и языковая информация. В этом случае все методы будут делать только то, что можно сделать на основе этой информации без знания машинной модели.

Эвристическая база данных содержит информацию о различных параметрах алгоритмов, которые пользователь сможет задавать через базу данных. Это также позволит проводить адаптивное обучение оптимизатора путем перебора параметров эвристик.

1.3 Самораспараллеливание

Одним из основных принципов построения блоков является самораспараллеливание – блоки организованы так, что каждую оптимизацию или алгоритм анализа запускают на масштабируемый регион транслируемого приложения. Они могут оптимизировать цикл, линейный участок, процедуру, модуль – различные участки приложения могут быть одновременно проанализированы и оптимизированы одними и теми же блоками (рис.2).

Здесь необходимо отметить, что использование методов и практик параллельного программирования делает продукты, произведенные на основе предлагаемой в данной работе технологии, частью строительных блоков индустрии параллельного программирования и в дальнейшем позволит существенно уменьшить время компиляции программ для многопроцессорных систем. Это в свою очередь даст возможность использования алгоритмически более сложных алгоритмов в целях оптимизирующей

компиляции, которые в настоящее время использовать не удается из-за ограничений на время компиляции. Это свойство автоматически распространяется на все продукты, которые будут построены с использованием предлагаемых инфраструктурных компонент.

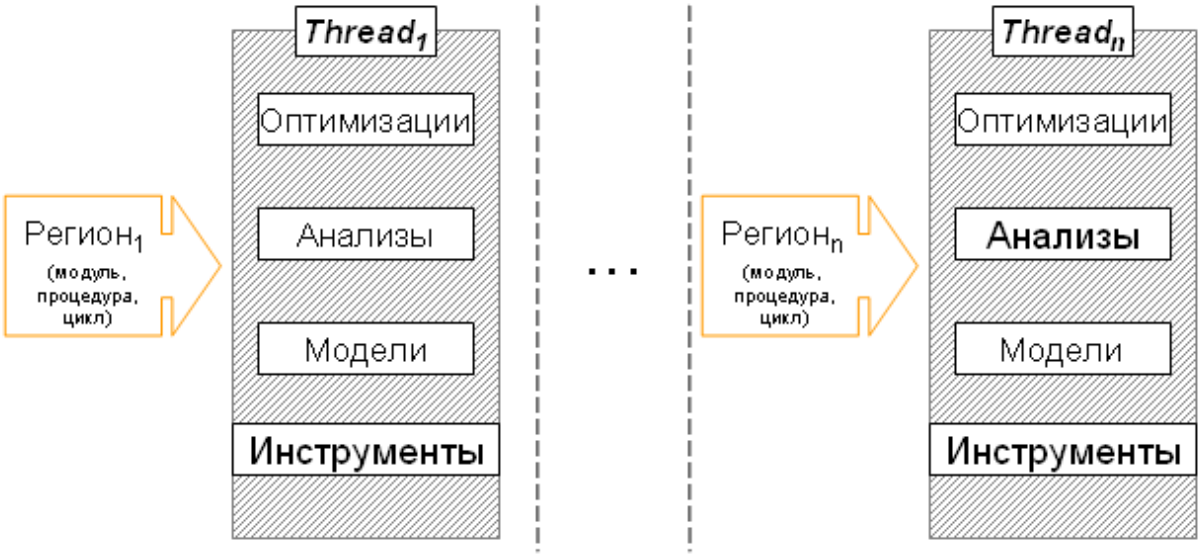

Рис. 2 Самораспараллеливание

1.4 Использование строительных блоков в продуктах

На основе предлагаемых блоков возможно создание как минимум трёх различных программных продуктов: анализатор, автоматический распараллеливатель и оптимизатор. Каждый программный продукт состоит из своего собственного набора компонент – подмножества полного набора компонент оптимизирующей компиляции. Подмноженства компонент, из которых состоит тот или иной программный продукт могут пересекаться частично или полностью в случае, когда один программный продукт включает в себя все компоненты другого программного продукта.

Анализатор включает в себя компоненты: инструменты, модели, анализы.
Автоматический Распараллеливатель включает все компоненты анализатора плюс компоненты, реализующие трансформации программы.
Оптимизирующий Компилятор включает в себя все компоненты автоматического распараллеливателя плюс компоненты планирования, распределения регистров, скалярных оптимизаций и другие.

2 Технология портирования блоков оптимизирующей компиляции

Современный оптимизирующий компилятор ([1-4, 6]) представляет из себя программный инструмент, с помощью которого можно откомпилировать программу, написанную на некотором входном языке программирования (C, C++, Fortran, binary, и. т. д.), в требуемый выходной язык. В процессе компиляции программа последовательно трансформируется в промежуточные представления, используемые компиляторами для сохранения семантики программы в процессе компиляции. На интерфейсах промежуточных представлений реализуются алгоритмы анализа и оптимизаций.

При создании каждой новой оптимизирующей системы разработчикам приходиться реализовывать заново алгоритмы на интерфейсах новых промежуточных представлений. Большинство известных коммерческих и открытых компиляторов используют одни и те же апробированные известные алгоритмы. Даже в рамках одного компилятора алгоритмы могут быть реализованы несколько раз на разных уровнях промежуточных представлений. Это многократное дублирование реализаций вызвано отсутствием в мире технологии унификации интерфейсов промежуточных представлений для возможности портирования алгоритмов анализа и оптимизаций. В данной работе предлагается метод, позволяющий отделить алгоритмы анализа и оптимизаций от промежуточных представлений и тем самым создать возможность для переиспользования их в любых компиляторных системах.

2.1 Методы портирования

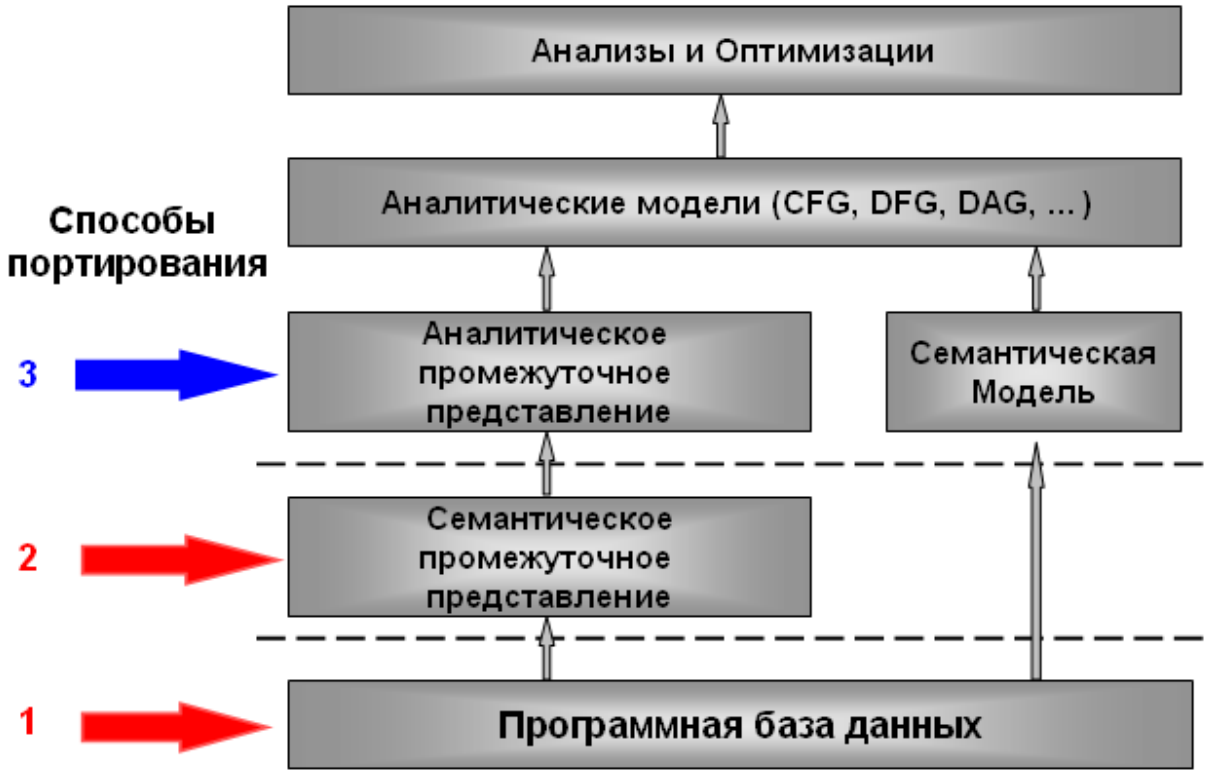

Рис. 3 Способы портирования

Существует три сценария взаимодействия блоков оптимизирующей компиляции с системой трансляции пользователя (рис. 3,4).

Первый сценарий: промежуточное представление пользователя представляется в виде файлового представления Базы Данных Программы, затем строится семантическое и аналитическое представления. После применения анализов и оптимизаций, семантическое представление преобразуется обратно в файловое представление, с которого происходит восстановление промежуточного представления пользователя.

Второй сценарий заключается в построении семантического представления из промежуточного представления пользователя без использования файлового представления, с последующим обратным построением пользовательского представления из семантического представления.

Третий сценарий: построение аналитического представления непосредственно на основе промежуточного представления пользователя, минуя семантическое представление.

Первый способ позволяет полностью отделить продукт пользователя от блоков оптимизирующей компиляции, что решает, в частности, вопросы лицензирования. Второй и третий сценарий предполагают линковку блоков непосредственно с системой пользователя. Эти способы делают работу конечной системы существенно более эффективной. Использование второй стратегии предполагается, в первую очередь, при создании систем компиляции полного цикла. Третья стратегия больше подойдет при использовании блоков в модернизации существующей системы со своим промежуточным представлением.

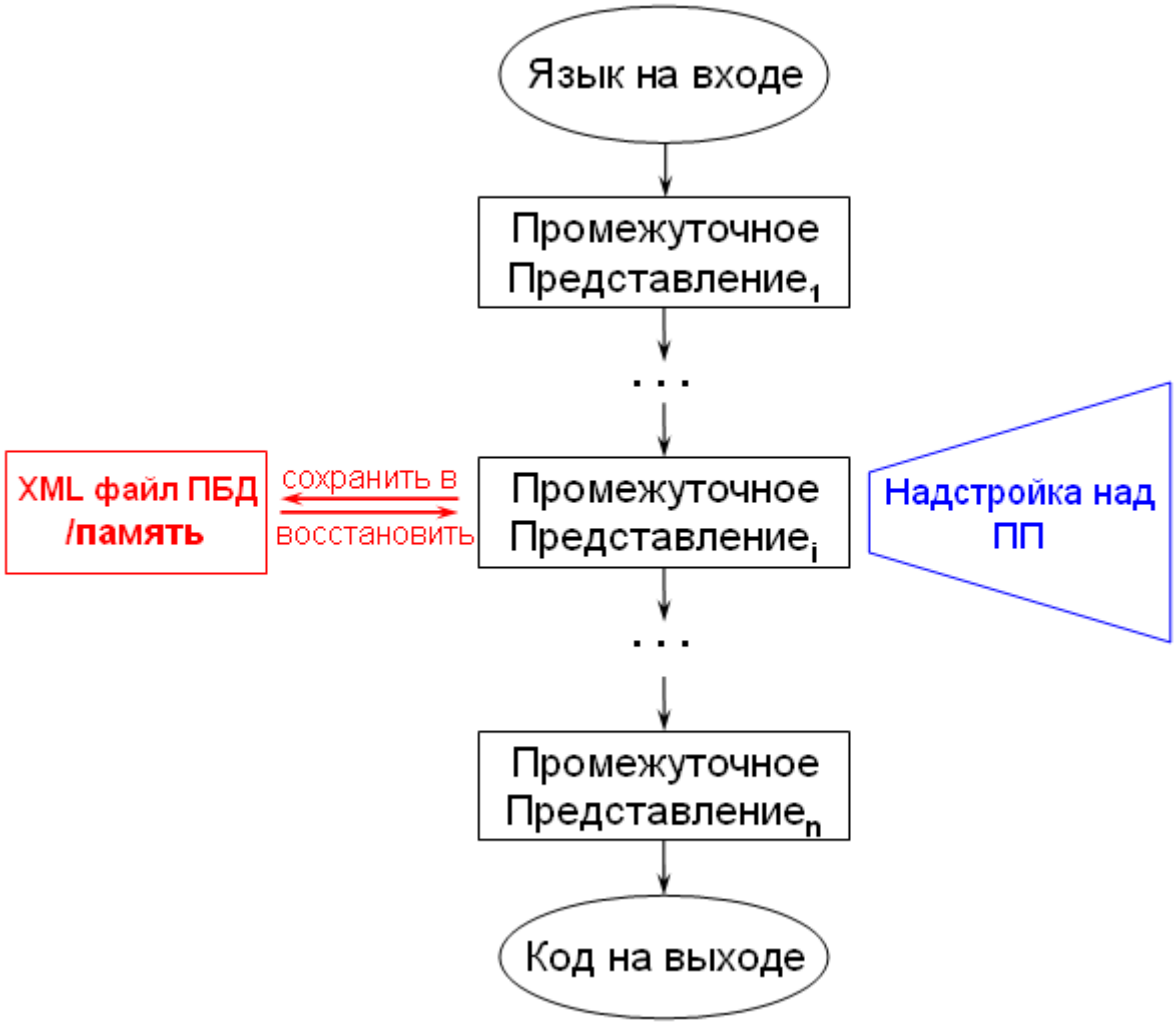

Рис. 4. Методика портирования

2.2 Семантическое представление

Любое промежуточное представление ([4]), используемое в компиляторах, в первую очередь сохраняет семантику исходной программы. Под семантикой программы понимается ее алгоритмическая сущность. С точки зрения фрагментации, программы, написанные на наиболее распространенных языках программирования (C, C++, Fortran и т.д.), организованы в модули и процедуры. Модуль соответствует файловой организации программ. В виде процедур оформляются фрагменты программ внутри модулей.

Структуры данных программ представляются объектами с фиксированными или динамически задаваемыми размерами и описанием их внутренней структуры с помощью типов. Наиболее общим способом сохранения алгоритмической составляющей программы в компиляторах является операционное представление (рис 5).

Операционное представление является списком операций. У операции может существовать входной контекст и выходной контекст. Входной и выходной контексты задаются списками аргументов и результатов. Аргументы могут быть литералами, объектами или ссылками на другие операции. Отображение связи между результатом одной операции и аргументом другой через объекты является более общей, чем представление этой связи в виде ссылки на операцию. Сама операция представляет собой набор атрибутов, определяющих семантическое действие над входным контекстом для получения выходного контекста. К такому операционному представлению может быть сведено практически любое известное промежуточное представление, начиная с синтаксических деревьев фронтендов (gcc, edg), заканчивая представлениями, наиболее приближенными к ассемблерам целевой архитектуры.

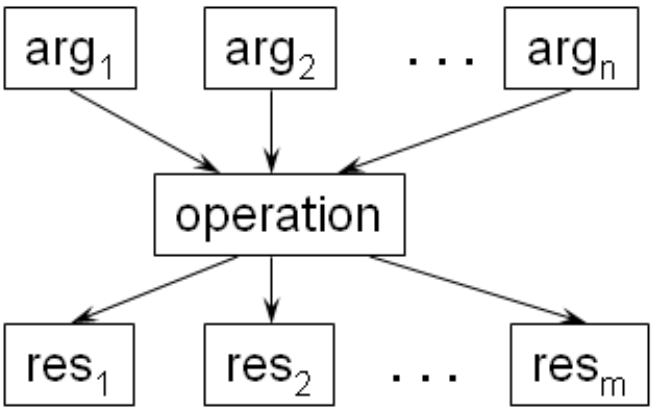

Рис. 5. Представление операции

2.3 Аналитическое представление

Одним из ключевых аспектов технологии портирования является отделение семантического представления программы от аналитических структур данных. Семантика программы должна быть полностью представлена в семантическом промежуточном представлении. Аналитические же структуры данных, такие как граф потока управления или граф потока данных строятся на аналитическом промежуточном представлении.

Для представления фрагментации программы на уровне аналитического представления вводиться понятие области видимости. Области видимости имеют иерархическую организацию. Каждый объект программы принадлежит одной из областей видимости. Понятия модуля и процедуры семантического представления программы представляются областями видимости в аналитическом представлении программы.

Элементами аналитического промежуточного представления являются аналитические операции (рис. 6). Входным контекстом для аналитических операций являются объекты и литералы. Выходным контекстом могут быть только объекты.

Аналитическое представление реализовано как надстройка над семантическим представлением. Аналитическая операция, аналитические объекты, аналитические литералы могут ссылаться на их аналог в семантическом представлении. Помимо элементов, имеющих аналоги в семантическом представлении, аналитическое представление содержит элементы, достраиваемые алгоритмами анализа. Например, при

построении графа потока данных достраиваются аналитические операции - φ-функции, которые не имеют аналогов в семантическом представлении.

Для связи аналитического представления и семантических представлений программы используется набор сервисов (интерфейсов), реализация которых зависит от конкретных семантических представлений. Суть сервисов состоит в том, чтобы унифицировать конкретные реализации семантических промежуточных для построения единого аналитического промежуточного представления и работы с ним. Связующие сервисы включают в себя обходы элементов семантического представления, получения атрибутов, изменения элементов и т.п.

Атрибуты элементов аналитического представления можно разбить на два класса. В первый класс попадают атрибуты, хранящиеся в элементах аналитического представления. Во второй класс попадают атрибуты, хранящиеся в элементах семантического представления и доступны через связующие сервисы семантического представления.

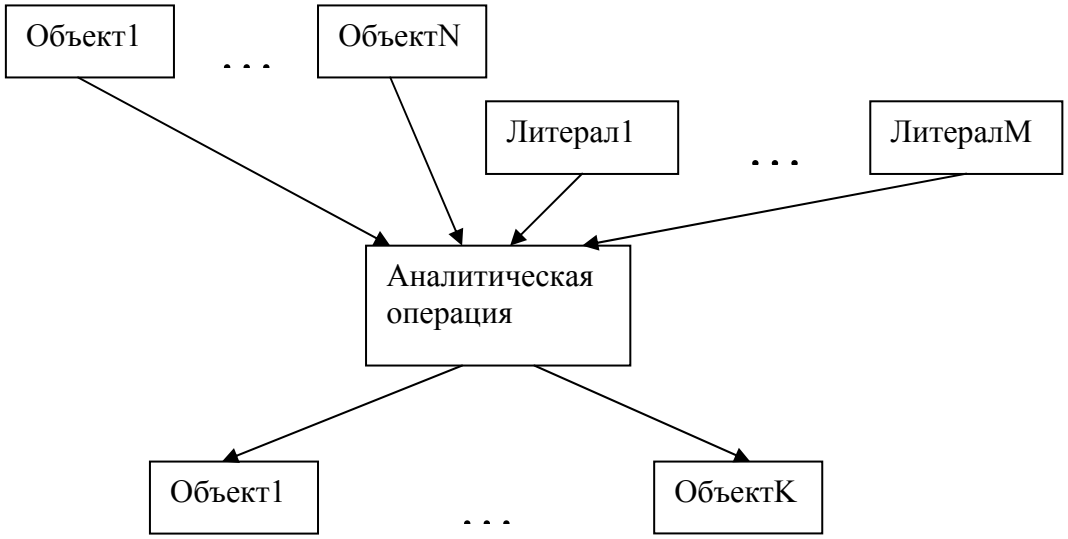

Рис. 6. Аналитичекая операция

2.4 Семантическая модель

Семантические представления программ, используемые в компиляторах, почти всегда отличаются набором семантических элементов (операций). В зависимости от близости семантических элементов к входным высокоуровневым языкам или выходным ассемблерам целевых архитектур представления называют высокоуровневыми и низкоуровневыми. Для того, что бы унифицировать работу с разного уровня представлениями в технологии портирования используется механизм описания семантики с помощью семантических моделей.

Семантическая модель представляет собой граф, состоящий из набора несвязанных подграфов. Каждый подграф описывает семантику операции над подмножеством входного контекста (аргументов) семантического элемента с изменением подмножества выходного контекста (результатов). Семантическая модель в целом описывает семантическую связь между аргументами и результатами операции семантического представления программы.

Узлами семантической модели являются семантические узлы с предопределенным количеством предшественников и преемников и с предопределенной смысловой (семантической) нагрузкой. Например, существует семантический узел, описывающий традиционное сложение с двумя аргументами и одним результатом.

Для ассоциации аргументов и результатов операций с входным и выходным контекстом семантической модели используются узлы входа и выхода.

На рис. 7 приведен пример семантической модели операции сложения с тремя аргументами.

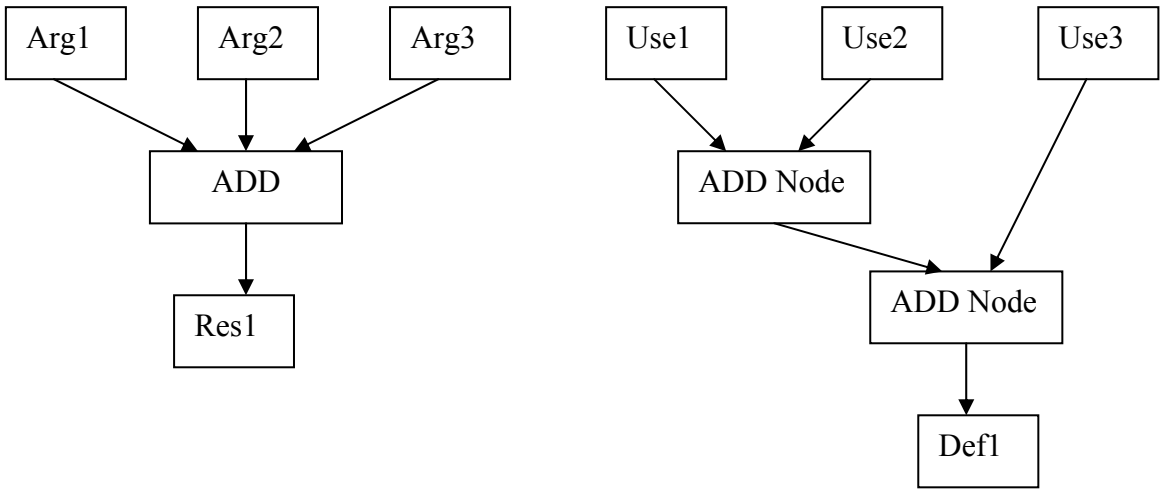

Рис. 7. Примет семантической модели

Механизм семантических моделей позволяет описывать семантические элементы пользовательского промежуточного представления через семантические элементы, понятные оптимизирующим блокам.

2.5 Совместимость с GCC

Технология, представленная в работе может применяться на платформах Windows, Linux или как кросс-платформенная. Исходный код транслируется с языков C, C++, Fortran компилятором GCC ([9]). С промежуточного представления GIMPLE семантика экспортируется в файл, из которого она далее разворачивается в промежуточное представление оптимизирующих блоков. На этом промежуточном представлении применяются оптимизации, после чего модифицированная семантика экспортируется обратно в файл, из которого она далее возвращается обратно в компилятор GCC, разворачивается в нем в его промежуточное представление и далее транслируется в код целевой машины, например x86, SPARC, PowerPC (рис.8).

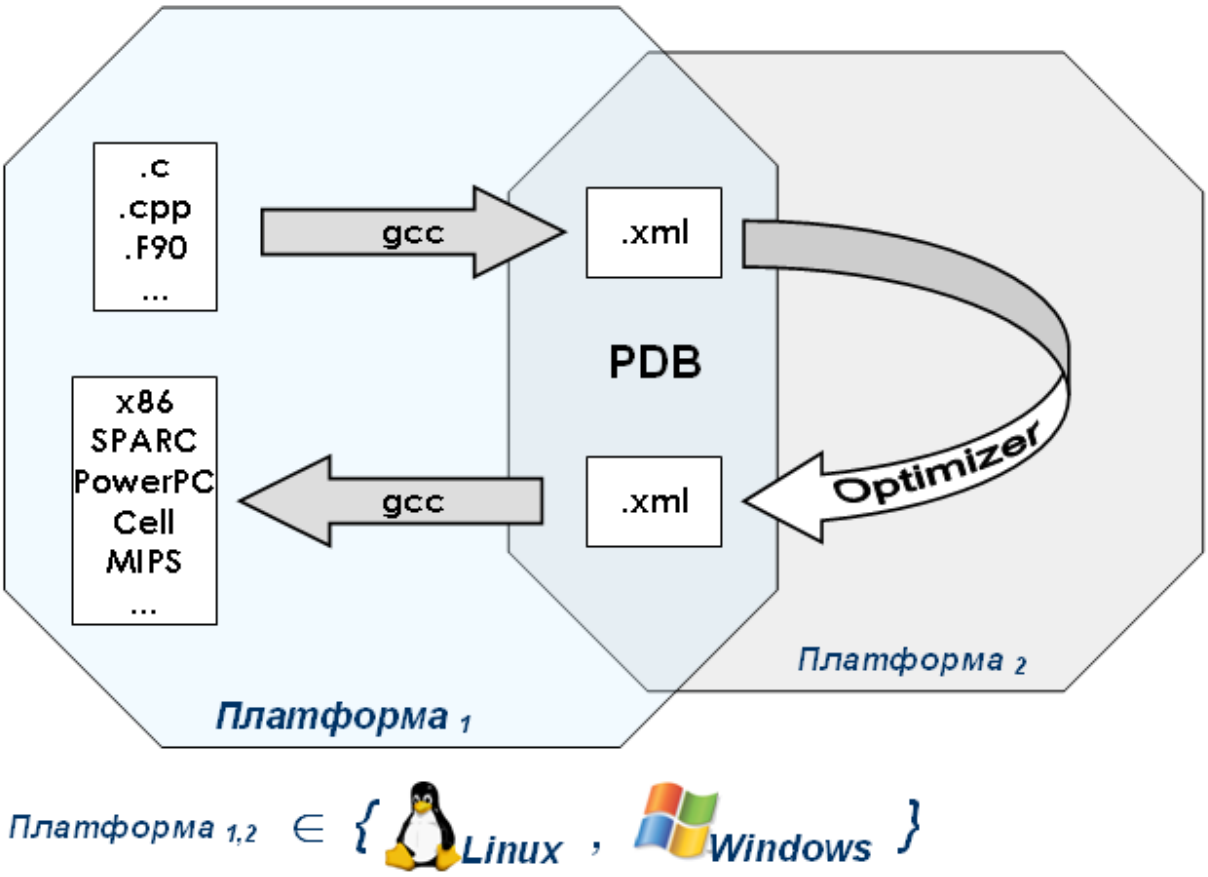

Рис. 8. Совместимость с GCC

3 Анализатор программ

Анализатор - это программа, которая выявляет параллельные вычисления, на основе использования методов статического анализа программ и включает в себя следующие блоки:

- Менеджер памяти
- Пакет Списков
- Пакет Битовых векторов
- Граф
- Дерево
- Пакет Хэш
- Пакет Векторов
- Пакет Стек
- Пакет Матриц
- Пакет XML парсер
- Пакет Маркер
- Пакет Решатель
- Пакет аналитических представлений
- Механизм временных атрибутов
- Пакет Строк
- Пакет Файл
- Сохранение-восстановление объектов с файла в память
- Граф потока управления

- Граф потока данных
- Граф зависимостей
- Анализ потока управления
- Анализ потока данных
- Анализ зависимостей в гнездах циклов
- Межпроцедурный анализ

3.1 Иерархия блоков оптимизирующей компиляции

Ниже, на рис. 9 изображена иерархия блоков оптимизирующей компиляции, разработанная в данной работе и которая является основой для реализации любой функциональности семейства продуктов, которые разработаны в ИТМиВТ.

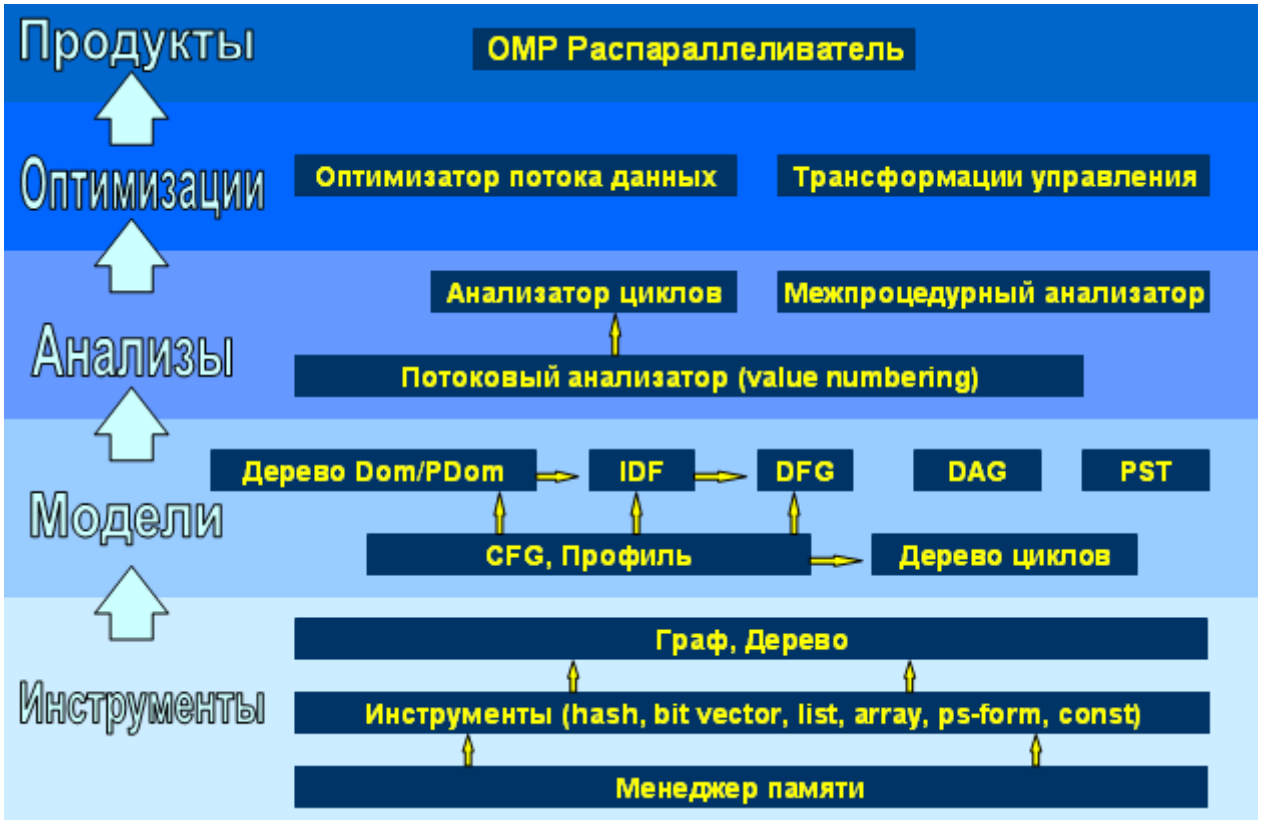

Рис. 9. Иерархия блоков оптимизирующей компиляции

3.2 Инструменты

Все контейнеры в инструментах обеспечивают безопасную работу с типами и совместимы с менеджером памяти. Контейнеры созданы на основе шаблонов C++, что позволяет достичь статической безопасности работы с типами.

Инструменты написаны в хост-независимым образом, что позволяет их быстро перенацеливать. В настоящее время поддерживаются две платформы – Win32/64 и Linux.

3.2.1 Менеджер памяти

Менеджер памяти контролирует работу с памятью. Может проверять корректность работы с памятью, собирать статистику, определять утечки. Испрользует надстройки в виде проверок, счетчиков и элементов сбора статистики над стандартными вызовами.

Свойства менеджера памяти:

- динамическая проверка типов в отладочном режиме – каждый доступ через поинтер проверяется на коректность соответствующего объекта, безопасные приведения типов и т.д
- проверка на утечки и выход за границы в отладочном режиме
- отсутствие отладочных проверок в рабочем режиме – все классы написаны способом, позволяющим компилятору подставлять процедуры в места вызова, некоторая функциональность отключена условной компиляцией
- полная поддержка семантики C++ в управлении памятью – классы и сервисы ведут себя почти так же, как поинтеры и операторы new/delete в C++ (за исключением некоторых краевых случаев в поиске методов)
- несколько типов пулов – позволяют точную настройку работы менеджера памяти, пулы оообъектов фиксированных типов, фиксированного размера, переменного размера
- возможность мгновенно удалить из пула все выделенные объекты, что предотвращает накопление утечек памяти из фазы в фазу
- обеспечивает каждый тред его собственными ресурсами и инфраструктуре менеджера памяти, что позволяет легко распараллеливать некоторые аспекты алгоритмов компиляции. Общий менеджер памяти охраняется своими собственными примитивами синхронизации

3.2.2 Пакет Списков

Обеспечивает три типа списков: двунаправленный список с указателями к первому и последнему элементам (т.е. граничные поинтеры), двунаправленный список без граничных поинтеров, однонаправленный список без граничных поинтеров.

3.2.3 Пакет Битовых векторов

Частный случай векторов, но имеющий более эффективную реализацию, обеспечивает несколько специфических методов работы с векторами как целыми объектами: OR, AND и т.д.

3.2.4 Граф

Набор универсальных интерфейсов для реализации произвольных графовых структур. Ключевая черта реализации заключается в добавлении графовых атрибутов к пользовательским объектам.

Свойства графа:

- Пространство дуг
- Маркирование узлов и дуг
- Нумерация – несколько нумраций для узлов одновременно, возможность быстро получить узел по нумерации.

Типы нумераций:

- идентификационная
- в глубину

- в ширину
- компактная
- пользовательская

Общие методы графа:

- проход графа в заданном направлении
- копировать или переместить узлы и дуги графа
- поиск сильно связанных компонент
- достижимость

Частным случаем графа является дерево. Для деревьев реализован набор универсальных инструментов для поддержки произвольных структур деревьев.

Свойства дерева:

- Маркирование узла
- Нумерация узлов
- Возможность прикрепить к узлам произвольные объекты

Общие свойства дерева:

- Присвоить уровни узлам
- Сортировать узлы по номерам

3.2.5 Пакет Хэш

Обеспечивает интерфейсы для создания и работы с так называемыми хэш таблицами (картами, наборами). Полностью конфигурируемый в смысле хэш функций и содержимых объектов. Имеется среда для вычисления хэш функций для известных типов, таких как интеграл, поинтер, и т.д.

3.2.6 Пакет Векторов

Массив объектов с проверкой границ, поддерживают ручной ресайзинг, C++-ная семантика оператора [ ].

3.2.7 Пакет Стек

Поддерживает “last in first out” порядок манипуляции объектов. Широко используется в алгоритмах компиляции. Может работать с несколькими контейнерами.

3.2.8 Пакет Матриц

Матрица объектов с несколькими определенными арифметическими операциями.

3.2.9 Пакет XML парсер

Поддерживает SAX интерфейс для разбора xml документов.

3.2.10 Пакет Маркер

Позволяет маркировать объекты в различных контейнерах: направленного графа, потокового графа, и т.д.

3.2.11 Пакет Решатель

Решает проблему набора афинных диофантиновых уравнений и неравенств. Воплощает симплексный метод.

3.2.12 Пакет аналитических представлений

Инструмент, нацеленный на хранение выражений, специфичный для оптимизирующих компиляторов. Выражения сохраняются в каноническом виде, представляющую собой сумму произведений.

3.2.13 Механизм временных атрибутов

Обеспечивает возможность присоединять объекты различных классов к существующим классам или отсоединять от них.

3.2.14 Пакет Строк

Обеспечивает “memory safe” интерфейсы работы со строками. Поддерживает строки в кодировках ASCII, MBCS, Unicode и операции с ними: конкатенация, поиск фрагмента и т.д.

3.2.15 Пакет Файл

Абстрагирует понятие файла и потока от конкретной операционной системы.

3.2.16 Сохранение-восстановление объектов с файла в память

Поддерживает автоматическую загрузку объектов в память или выгрузку из неё. Позволяет работать с большими структурами – например структурами межпроцедурного анализа.

3.3 Модели:

3.3.1 Граф потока управления

Управляющий граф процедуры ([1]) является аналитической структурой данных, отражением результатов анализа топологии и семантики программы. Каждый узел управляющего графа соответствует некоторому линейному участку. Управляющий граф является ориентированным. Каждая дуга такого графа соответствует возможности передачи управления в программе между линейными участками.

Линейным участком называется упорядоченное множество операций. Если множество не пусто, то выделяются входная и выходная операции и выдвигаются требования, которым должен удовлетворять линейный участок:

- **связность**: все операции линейного участка, включая конечную, достижимы из начальной операции без выхода за пределы линейного участка; из всех операций линейного участка, включая начальную, достижима конечная операция
- **замкнутость**: выход из линейного участка минуя конечную операцию или вход в линейный участок минуя начальную операцию невозможен

- **полнота**: линейному участку принадлежат все операции проходимые при обходе сверху от начальной операции к конечной и при обходе снизу от конечной операции к начальной

- **одноуровневость**: линейный участок не содержит внутри себя других линейных участков

- **ацикличность**: ни одна операция линейного участка не может быть достигнута при проходе по управлению от самой себя без выхода за пределы линейного участка

Все узлы графа делятся на два типа: обычные узлы и узлы слияния. Обычный узел может иметь не более одного предшественника. Узлы слияния могут иметь произвольное число предшественников.

Один узел графа помечен как стартовый, такой узел не имеет входных дуг. Стартовый узел соответствует линейному участку, с которого начинается выполнение процедуры. Один узел графа помечен как стоповый, в него сведены все дуги, по которым может происходить выход из процедуры.

Граф может содержать циклы. Представим, что мы обходим граф "вширь" помечая уже рассмотренные узлы. В процессе такого обхода могут встретиться дуги, ведущие к уже помеченному узлу. Такие дуги будем называть обратными (рис. 10).

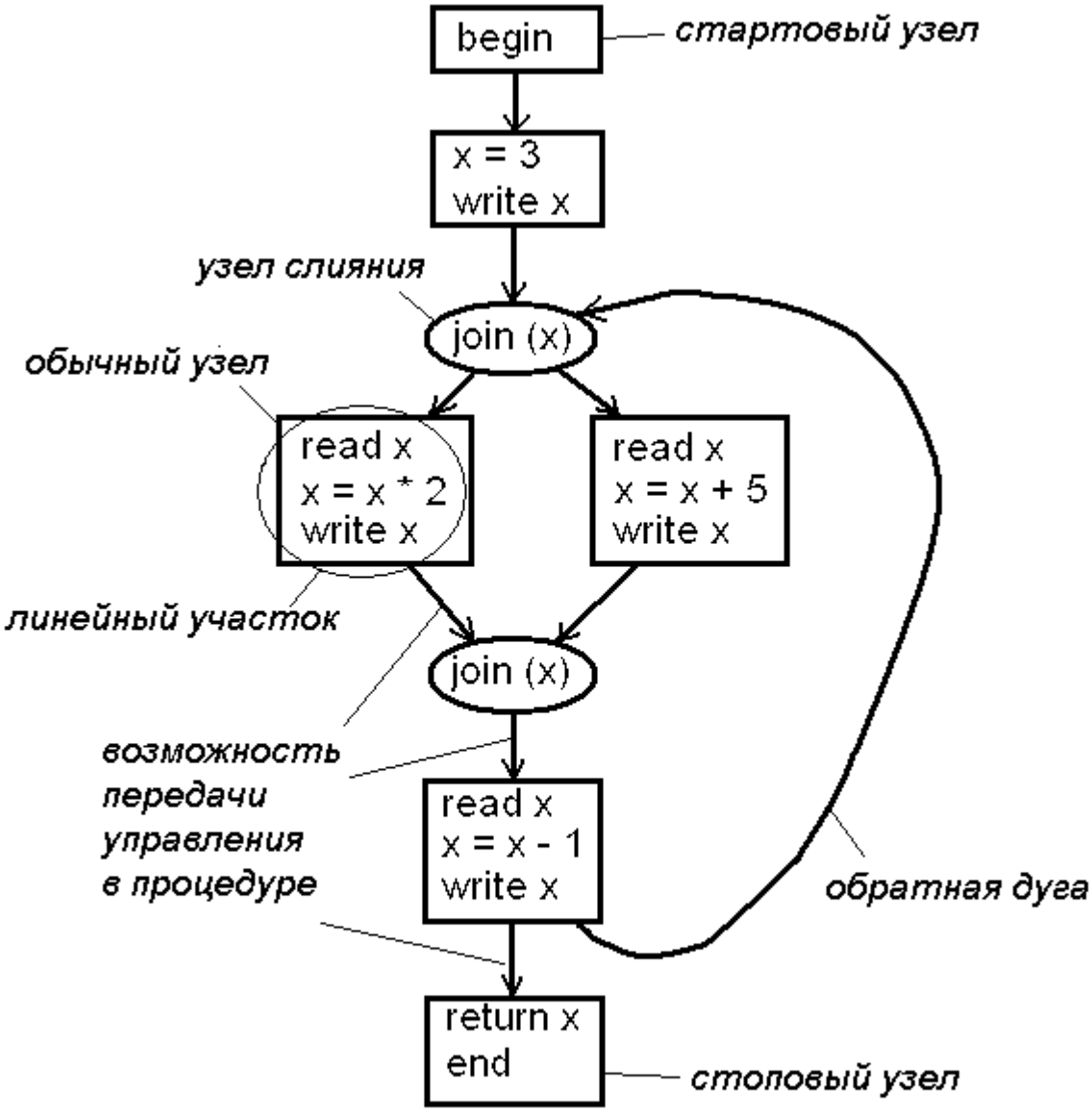

Рис. 10. Управляющий граф

3.3.2 Граф потока данных

При потоковом анализе крайне полезным оказывается граф потока данных ([4, 6]), который связывает между собой результаты и аргументы операций. Граф потока данных позволяет для аргументов получать операции, вырабатывающие значение аргумента. Кроме графа потока данных крайне полезной для анализа является форма единственного присваивания. Программа, переведенная в эту форму, содержит псевдооперации в точках схождения управления. В результате с учетом этих псевдоопераций, для каждого аргумента, для которого произведен перевод в форму единственного присваивания, существует единственная запись. В графе потока данных для таких аргументов будет всего одна входная дуга от единственной записи. На рис. 11 показан граф потока данных в форме единственного присваивания. В программе, представленной на рис. 11 есть две записи в переменную A и одно чтение. Запись обозначается как ‘A = …’, чтение обозначается как ‘… = A’. Прямоугольниками и связывающими их стрелками отображается поток управления в программе. Из блока BLOCK 1 возможна передача управления на блок BLOCK 2 или на блок BLOCK 3. Из блоков BLOCK 2 и BLOCK 3 управление переходит в блок BLOCK 4. Черными кружками и стрелками, соединяющими их, нарисован граф потока данных для данного фрагмента программы. На графе потока данных каждый узел соответствует операции, и каждая дуга соответствует паре аргумент-результат. Псевдооперация, соответствующая схождению потока управления, обозначена как ‘φ(A)’.

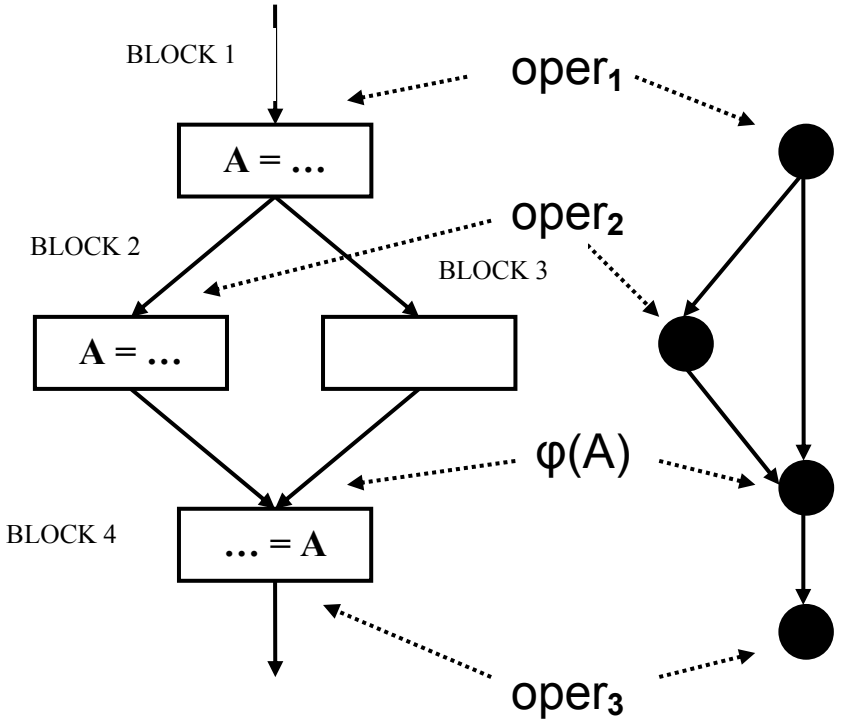

Рис. 11. Граф потока данных в форме единственного присваивания

3.3.3 Ациклический граф зависимостей

Зависимости в программе ([1-6]) могут быть представлены в виде ациклического графа (рис. 12). Граф зависимостей представляет собой ориентированный связный граф без циклов. Узлами этого графа являются операции. Дугами графа являются зависимости.

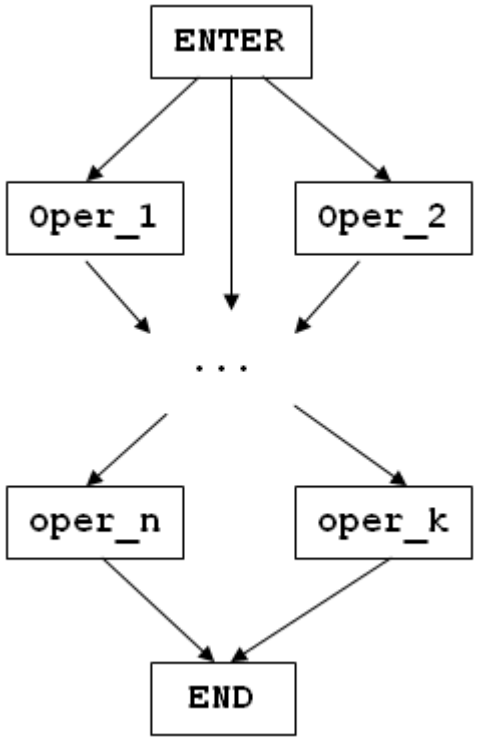

Рис. 12. Ациклический граф зависимостей

3.3.4 Call Graph (Граф вызовов)

Граф вызовов [4, 6] отражает отношение вызываемый-вызывающий на множестве процедур программы. Узлами графа вызовов являются процедуры программы. Если процедура a вызывает из себя процедуру b то в графе вызовов процедур будет построена дуга a→b.

3.4 Методы статического анализа программ

Основная задача, которую решают методы статического анализа программ в оптимизирующих компиляторах – это определение отношения зависимости по данным и управлению между различными группами вычислений программы [1-6]. Особенно важное значение методы статического анализа программ играют при построении анализатора и автоматического распараллеливателя, для обнаружения независимых участков программы (анализатор) и их дальнейшего оформления в виде параллельных блоков (распараллеливатель).

3.4.1 Анализ потока управления

Под анализом потока управления понимается совокупность методов исследования топологии управляющей структуры программы, рассматриваемой независимо от остальной семантики программы. Предметом исследования здесь является граф управления и его свойства, определяемые его топологией [1-6]. Перечислим основные понятия и отношения, вычисляемые на основе анализа потока управления:

- отношение доминирования;
- отношение постдоминирования;
- отношение зависимости по управлению;
- отношение достижимости;
- фронт доминирования;
- итерационный фронт доминирования;
- сильно связная компонента;

и др.

Для компактности представления и ускорения запросов многие из перечисленных понятий представляются в виде самостоятельных структур данных с различными топологическими свойствами: дерево доминаторов, дерево постдоминаторов, матрица достижимости и др.

3.4.2 Анализ потока данных

В процессе анализа потока данных моделируется исполнение программы над всеми значениями из некоторой области ее реальных данных, представленных в символическом виде. Другими словами, задача потокового анализа состоит в построении определенного вида утверждений о реальном поведении программы в некоторых специально выделенных точках, которые верны для всех возможных исполнений программы при любом наборе входных данных из рассматриваемой области. Такие задачи решаются методом *глобального потокового анализа* или *абстрактной интерпретации* [8].

Потоковый анализ естественным образом разделяется на внутрипроцедурный и межпроцедурный уровни.

3.4.3 Межпроцедурный анализ

Решение задачи межпроцедурного анализа потока данных ([6]) имеет первостепенное значение для нахождения параллелизма, как на уровне команд, так и функционального параллелизма (ациклические фрагменты, которые могут работать параллельно). Это позволяет параллельно исполнять команды, целые процедуры и отдельные регионы.

Ключевыми параметрами в постановке задачи межпроцедурного анализа являются:

- множество переменных (объектов) программы, для которых проводится анализ;
- степень детализации результатов анализа.

3.4.3.1 Множество объектов программы

Все объекты программы можно разделить на два класса:

1. Статические объекты, которые пользователь явно определил статически, и память под которые отводиться во время компиляции. Они составляют символьную таблицу программы ([1]).

2. Объекты, память под которые заказывается в процессе исполнения программы. Эти объекты иногда называют динамически заказываемыми объектами или heap-объектами.

Статические объекты делятся на скалярные, структурные и объекты-массивы.

Как правило, для статических объектов структурного типа различают поля структур и не различают отдельные элементы объектов-массивов.

Для динамических объектов, как правило, выбирают одну из следующих стратегий аппроксимации:

1. Все динамические объекты рассматриваются как единый объект.
2. Разбивают на классы эквивалентности по части статической цепочки вызовов.
3. Разбивают на классы эквивалентности по размеру заказываемого объекта.
4. Разбивают на классы эквивалентности по способу доступа к таким объектам (k-limiting).

3.4.3.2 Степень детализации результатов анализа

Степень детализации результатов анализа имеет два измерения ([6, 7]):

1. Внутрипроцедурное (flow sensitivity);
2. Межпроцедурное (context sensitivity).

Внутрипроцедурная детализация определяет элемент программы, для которого вычисляются индивидуальные результаты анализа. Таким элементом может служить отдельная операция (максимальная степень детализации), линейный участок, цикл, сильно-связная компонента графа управления, вся процедура (полное отсутствие детализации).

Межпроцедурная детализация определяет, как мы оцениваем вклад конкретного вызова процедуры в результаты анализа. Фактически это определяет, каким образом мы различаем результаты анализа для одной и той же процедуры в зависимости от места ее вызова и отличаем ли вообще (полное отсутствие детализации).

3.4.4 Анализ зависимостей в гнездах циклов

Анализ зависимостей в гнездах циклов один из наиболее ранних и хорошо изученных подходов к нахождению параллелизма в программе [5]. Как уже отмечалось, в цикловых регионах все операции взаимно достижимы и поэтому традиционное понятие зависимости по данным тривиально сводится к понятию конфликт. Дабы избежать этой трудности, вводится отношение порядка на пространстве итераций и тогда определение зависимости по данным снова обретает понятный смысл.

Рассмотрим постановку задачи:

Дано гнездо циклов:

```
  for i1 = p1,q1
    for i2 = p2,q2
      . . . . . .
      for in = pn,qn
      {
        M[a1]...[am] = M[b1]...[bm];
      }
```

где p,q,ai,bi - линейные функции от переменных i1,...,in.

Зависимость на пространстве определяется следующим образом: если на двух разных итерациях I и J (I и J - векторы размера n состоящие из значений i1,...,in на итерации) происходит обращение, чтение или запись, к одной и той же ячейке памяти, причем итерация с номером I выполняется раньше итерации с номером J и между I и J нет обращений в это место памяти, то мы говорим, что между итерациями I и J есть зависимость.

4 Автоматический распараллеливатель

Автоматический распараллеливатель (АР) – это программа, которая утилизируют информацию о независимых участках вычислений в программе с помощью каких-либо инструментов параллельного программирования ([6]). Представленный в данной работе автоматический распараллеливатель имеет следующие основные технологические черты:

- Распараллеливает приложения для многоядерных архитектур
- Используте результаты Анализатора программ, который вычисляет независимые участки программ
- Распараллеливает для произвольной аппаратной платформы
- Реализован на основе технологии Инструментов и Строительных Блоков, представленных в данной работе.

Автоматический распараллеливатель отображает информацию о параллельности в формат OMP ([10]). Он реализован на базе GCC ([9]) с двумя возможными режимами работы:

a) C, C++, Fortran -> GCC -> AP(OMP) -> GCC -> X86
b) C++ -> GCC ->AP (RM) -> GCC -> C++

Продукт работает как на платформе Windows так и на платформе Linux, а также и в кросс платформенном варианте.

Автоматический распараллеливатель включает в себя следующие продукты и блоки:

- Анализатор
- Скалярный оптимизатор
- Высокоуровневый оптимизатор
- Распараллеливатель циклов и секций в формат OMP ([10])
- Векторизатор

4.1 Результаты по производительности автоматического распараллеливателя программ

Автоматический Распараллеливатель показал впечатляющие результаты на 6 задачах из пакета CPU2006. Производилось сравнение автораспараллеливающей системы с наиболее эффективного компилятора для x86 платформы - icc 10.1.008. Для этого использовалась машина: 2 x Intel Xeon X5365 3.0GHz, 4 core, FSB 1333MHz, 32+32 Kb L1 cache, 4+4Mb L2 cache, 4 Gb DDR2 со следующими флагами компиляции (скачать альфа-версию распараллеливателя можно с сайта ([13])):

icc	icc -O2 -ipo -static -no-prec-div
icc parallel	icc -O2 -parallel -ipo -static -no-prec-div
gcc	gcc -O2
uopt-gcc	gcc -O2 + auto-parallelizer [с опциями]*

* - опции запуска

459.GemsFDTD	--inter-module
462.libquantum	--inter-module -pto
470.lbm	--inter-module -pto
все остальные	без опций

Результаты сравнения приведены ниже (Табл. 1). Время выражено в секундах. Поле ratio показывает отношение времени работы программы на системе, описанной в начале данного раздела, к времени исполнения на эталонной машине ([12]):

Результаты на тестах из пакета Spec2006 Таблица 1

Benchmark	icc		icc -parallel		gcc		uopt-gcc	
	time	ratio	time	ratio	time	ratio	time	ratio
410.bwaves*	716	18.98	509	26.69	988	13.76	482	28.19
436.cactusADM	1421	8.4	222	53.82	1803	6.63	332	35.99
437.leslie3d	887	10.59	681	13.8	1129	8.33	449	20.93
459.GemsFDTD	1030	10.3	669	15.85	1402	7.57	845	11.82
462.libquantum	1340	15.4	1340	15.4	1390	14.9	643	31.34
470.lbm	2022	6.79	2030	6.76	2241	6.13	589	22.86

На основании числовых величин из таблицы на рис.13 результаты представлены в графическом виде.

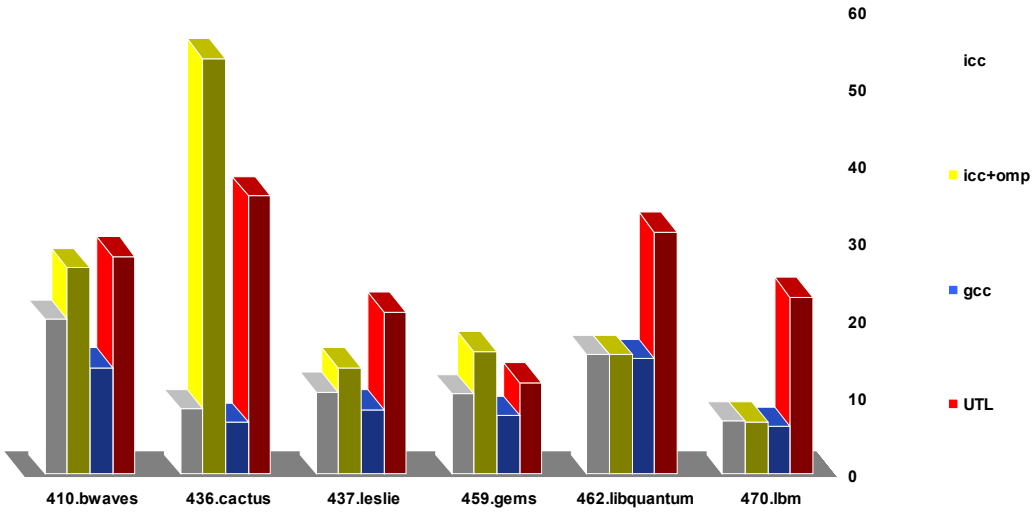

Рис. 13. Сравнительные результаты производительности

Ощутимый прирост производительности при распараллеливания компилятором icc наблюдался только на 4х тестах - 410.bwaves, 436.cactus, 437.leslie, 459.gems. Они анализировались в первую очередь и на них автоматический распараллеливатель также показал великолепные результаты производительности, на двух из них, 410.bwaves и 437.leslie превзойдя по производительности icc.

На тесте 470.lbm и 462.libquantum icc не получил никакого увеличения производитеьности от распараллеливания, в то время как распараллеливатель, разработанный на основе технологии блоков оптимизирующей компиляции, показал

почти 3-х и 2-х кратное увеличение производительности от автоматического распараллеливания на данных задачах.

Ниже на рис. 14. приведены абсолютные приросты производительности от применения автораспараллеливателей и от применения оптимизирующих компиляторов. Чем длиннее прямоугольник, соответствующий определенному компилятору, тем больший вклад он вносит в прирост производительности на данной задаче.

Основной вывод, который можно сделать на основе данного сравнения – для многоядерных архитектур для данного набора задач прирост производетельнсти от оптимизации для одного ядра исчезающе мал по сравнению с тем вкладом, который дает автоматическое распараллеливание.

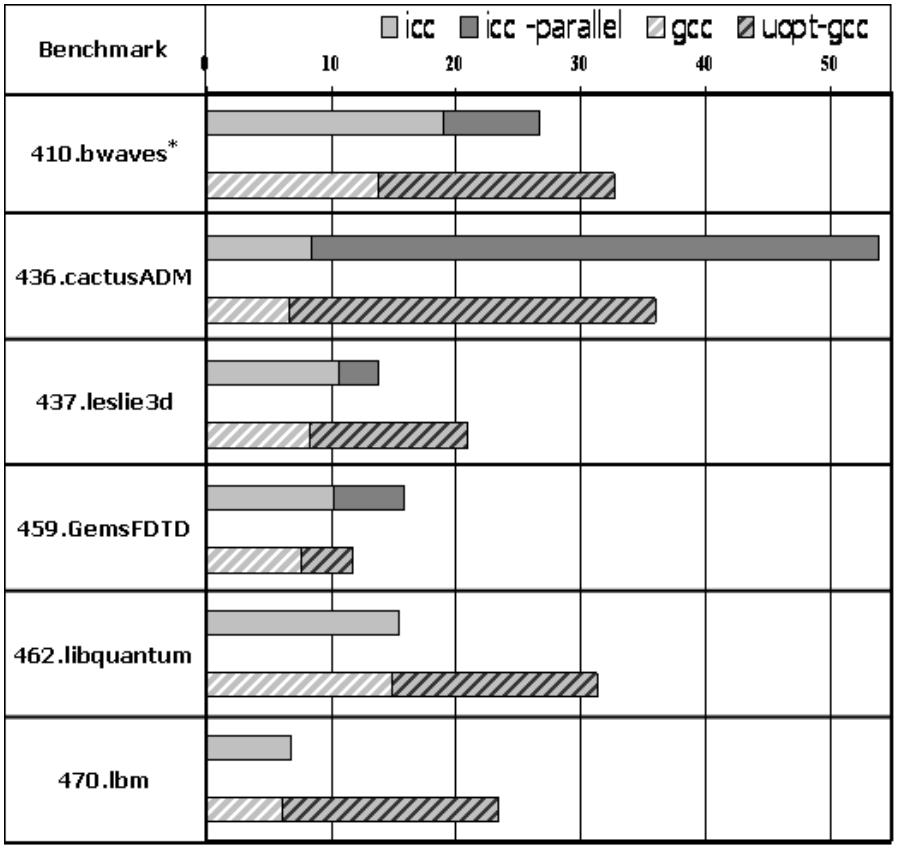

Рис.14. Абсолютный прирост производительности от распараллеливателей и компиляторов

5 Заключение

Перечисленные выше технологии были с успехом применены для создания 2 продуктов - *анализатора программ* и *автоматического распараллеливателя*. Альфа версии этих продуктов выложены в Интернет с возможностью скачивания и апробации для возможных потребителей ([13]).

Автоматический распараллеливатель, встроенный в GCC ([9]), показал свое превосходство над одним из лучших на сегодняший день автоматических распараллеливаетелей – компилятором компании Intel – icc ([11]).

Разработанную в ИТМиВТ технологию оптимизирующей компиляции можно в перспективе использовать в качестве основы построения оптимизирующих

компиляторов и смежных программных продуктов, как в оборонной, так и в гражданской информационной индустрии России.

6 Список литературы

1. **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, Reading, 1986.
2. **Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen**, Modern Compiler Design, by John Wiley & Sons,Ltd, 2000.
3. **Randy Allen, Ken Kennedy**, Optimizing Compilers for Modern Architectures. 2002 by Academic Press.
4. **Steven S. Muchnick**, “Advanced Compiler Design and Implementation”, Morgan Kauffman, San Francisco, 1997.
5. **Utpal Banerjee**, Loop Transformations for Restructuring Compilers. Kluwer academic Publishers, 1993.
6. **Альфред В. Ахо, Моника С.Лам, Равви Сети, Джеффри Д. Ульман,** Компиляторы: принципы, технологии и инструментарий, Издательский дом «Вильямс», Москва - Санкт-Петербург – Киев, 2008
7. **Maryam Emami** “A practical interprocedural alias analysis for optimizing/parallelizing C compiler”. Master’s thesis, School of Computer Science, McGill University, August 1993. 19
8. **P. Cousot, R. Cousot**, “Abstract interpretation framework”. Journal of logic and Computation, 2(4) 511-547, August 1992.
9. www.gnu.org
10. www.openmp.org
11. www.intel.com
12. www.spec.org
13. www.optimitech.com