

Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости

В.Ю. Волконский

Аннотация. Работа посвящена методам оптимизации, используемым в современных компиляторах для архитектур с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости. В ней дается сравнительный анализ аппаратных и программных методов оптимизации и обеспечения совместимости программ, а также краткое описание базовой архитектуры, поддерживающей компиляторные методы оптимизации и эффективной двоичной совместимости. Рассматриваются важнейшие методы оптимизации программ, введенные для базовой архитектуры. Они включают в себя методы анализа зависимостей, методы оптимизации применительно к платформе с явным параллелизмом команд, методы оптимизации обращений в память, методы адаптивной оптимизации, быстрые алгоритмы оптимизации, особенно важные для динамических оптимизирующих компиляторов. Рассмотрены структурные изменения в оптимизирующих компиляторах и технология их отладки.

Введение

При создании современного программного обеспечения применяются два доминирующих подхода:

1. Программы пишутся на языках высокого уровня, таких как C, C++, Fortran и других, а после компиляции распространяются в виде двоичных кодов конкретных *аппаратных* платформ.
2. Программы пишутся на переносимых языках высокого уровня, таких как Java или C#, и распространяются после компиляции в виде кодов *виртуальных* платформ.

Первый подход ставит перед разработчиками программного обеспечения трудную проблему адаптации программ к различным аппаратным платформам. Прямым следствием этих трудностей становится сокращение числа поддерживаемых платформ, что в конечном итоге приводит к доминированию одной аппаратной платформы. Именно эту тенденцию мы наблюдаем на примере платформы Intel x86 (IA-32). Второй подход оказывается гораздо более привлекательным для разработчиков программного обеспечения, поскольку в этом случае задача адаптации передается тем, кто реализует виртуальные платформы на конкретных аппаратных платформах.¹

Однако выбор языка реализации и платформы определяется не только удобством распространения программ, но и требованиями к их производительности. Значительная часть создаваемого программного обеспечения требует высокой производительности, поэтому вопросам оптимизации исполнения программ уделяется повышенное внимание. При этом результат достигается двумя

¹ Существует еще один способ распространения программного обеспечения – это, так называемые, программы с открытым кодом. Программы этого класса распространяются в форме текстов на языках высокого уровня. Но перед исполнением они преобразуются в коды конкретных аппаратных или виртуальных платформ, так что этот способ распространения не вносит ничего нового по отношению к указанным доминирующим подходам.

способами: использованием оптимизирующих компиляторов и аппаратными средствами. Их независимое применение часто приводит к необходимости повторно выполнять одни и те же оптимизации, что снижает эффект от применения программных способов, с одной стороны, и усложняет аппаратуру – с другой. Поэтому активно ведется поиск аппаратных решений, при которых функции оптимизации и обеспечения совместимости были бы целиком переданы оптимизирующим компиляторам.

В данной работе рассматриваются методы реализации оптимизирующих компиляторов для микропроцессора Эльбрус-3М, архитектуре которого свойственен явный параллелизм команд и аппаратная поддержка двоичной совместимости с наиболее распространенной аппаратной платформой IA-32 [1]. В дальнейшем она будет обозначаться как «базовая». Эти методы нашли свое воплощение в реализациях оптимизирующих компиляторов с языков C, C++ и Fortran-90 (в дальнейшем ОКЭ) и оптимизирующей системы двоичной трансляции кода IA-32 (в дальнейшем ОДТЭ) для базовой архитектуры. Рассматриваемый в работе программно-аппаратный подход к оптимизации позволяет эффективно исполнять программное обеспечение, создаваемое и распространяемое с помощью любого из указанных выше подходов.

Статья содержит следующие разделы. В *первом* разделе дается сравнительный анализ аппаратных и программных методов оптимизации и обеспечения совместимости программ. В него включено также краткое описание архитектурных особенностей, поддерживаемых рассматриваемые в дальнейшем компиляторные методы оптимизации и обеспечения эффективной совместимости. Во *втором* разделе рассматриваются важнейшие методы оптимизации программ, введенные для базовой архитектуры. Они включают в себя методы анализа зависимостей, методы оптимизации применительно к платформе с явным параллелизмом команд, методы оптимизации обращений в память, методы адаптивной оптимизации, быстрые алгоритмы оптимизации, особенно важные для динамических оптимизирующих компиляторов, а также связанные с этим структурные изменения в компиляторах. В *третьем* разделе рассматриваются методы отладки, используемые при разработке таких сложных программ, как ОКЭ и ОДТЭ. В *заключении* подводятся итоги достигнутых результатов и обсуждаются направления дальнейших исследований.

1. Аппаратные и программные методы оптимизации

При независимом выполнении оптимизации аппаратными и программными методами аппаратура также обеспечивает совместимость с существующими кодами исходной платформы, что требует введения в нее дополнительных средств преобразования (компиляции) двоичных кодов. При таком подходе многие одинаковые оптимизирующие функции зачастую выполняются и в аппаратуре, и в компиляторах. Но возможен другой подход к оптимизации программ с сохранением полной совместимости без аппаратного выполнения старых двоичных кодов. Он предполагает включение в аппаратуру таких средств, которые поддерживают реализацию оптимизаций и обеспечение совместимости чисто программными методами на базе оптимизирующих языковых компиляторов и двоичных трансляторов. Этот подход позволяет развивать и упрочнять аппаратную архитектуру, сохраняя при этом полную программную совместимость существующих двоичных программ. Оба подхода рассматриваются в данном разделе.

1.1. Сходства и различия в аппаратных и программных методах оптимизации

Рассмотрим, методы оптимизации, используемые в компиляторах и на аппаратном уровне. Цели оптимизации в обоих случаях близки, хотя существенно различаются возможности. Первая цель состоит в том, чтобы избавиться от лишних вычислений, вторая – наиболее оптимально адаптировать программу к имеющимся архитектурным особенностям и аппаратным ресурсам. При этом обе цели не являются полностью независимыми. Иногда адаптация к аппаратным особенно

стям требует увеличить число выполняемых команд, чтобы сократить общее время их выполнения. Но это возможно только при наличии параллельных аппаратных ресурсов.

Программа, поступающая на вход оптимизирующего компилятора, предварительно преобразуется в вид, удобный для анализа и оптимизаций. Это справедливо как для языковых, так и для двоичных оптимизаторов. Далее для нее строятся специальные вспомогательные структуры данных, облегчающие оба процесса. При анализе компилятор должен найти и устранить ложные зависимости, для чего зачастую используется техника переименования регистров, а при оптимизации должен избавиться от лишних вычислений, найти наиболее часто выполняемые последовательности команд, опираясь на профильную информацию, и построить для них оптимальный код, включая планирование для архитектур с параллельными исполняющими устройствами. Конечно, это несколько общий взгляд на функции, выполняемые современным оптимизирующим компилятором, но даже его достаточно, чтобы увидеть сходство между методами аппаратной и программной оптимизации.

Современные микропроцессоры, например, микропроцессоры платформы IA-32 [2], сначала аппаратно декодируют сложные команды переменной длины и преобразуют их в более простые и регулярные микрооперации. Далее выполняется переименование регистров, чтобы исключить ложные зависимости между микрооперациями, обусловленные ограниченным количеством регистров в исходной системе команд. При этом выполняются некоторые оптимизации, в частности, из командного потока исключаются операции чтения из памяти, если в этом потоке им предшествуют записи по тому же адресу. Затем формируется трасса перекодированных микроопераций, которая представляет собой наиболее вероятную цепочку операций не с одного, а с нескольких следующих один за другим линейных участков исполнения кода. Эта трасса помещается в специальную скрытую память (кэш трасс) для повторного использования. Чтобы обеспечить наиболее оптимальный набор трасс, аппаратно поддерживается специальная обучающая система, которая наблюдает за выполнением операций передачи управления в программе и стремится предсказать направление перехода в каждой точке. Наконец, аппаратура выполняет планирование выполнения микроопераций на заданном парке имеющихся исполняющих устройств. Для микропроцессоров с упрощенной архитектурой (RISC) удается избавиться только от фазы перекодировки операций, но все остальные действия выполняются аппаратно [3, 7].

Существует и несколько важных отличий между оптимизацией в аппаратуре и в компиляторах. Во-первых, традиционные языковые компиляторы являются статическими, в то время как аппаратные компиляторы по сути своей динамические. Во-вторых, существенно различается объем информации о программе, используемый в этих двух типах компиляторов. Языковый компилятор знает много дополнительных свойств программы, которые недоступны аппаратному оптимизатору, поскольку они не сохраняются в двоичном коде. Наконец, языковый оптимизатор может использовать для анализа и оптимизации существенно более крупные регионы программы, в то время как возможности аппаратного оптимизатора ограничены несколькими десятками команд трассы исполнения программы.

Что же мешает разработчикам новых поколений микропроцессоров избавиться от той части функций, которая вполне успешно и зачастую более эффективно может быть реализована в оптимизирующем компиляторе? Это проблема совместимости. Поскольку все программы распространяются в виде двоичных кодов, единожды зафиксированная система команд становится тем постоянно действующим интерфейсом, изменения которого могут быть сделаны только в направлении расширения его функциональности с полным сохранением всех ранее заложенных функций и способов их кодирования. При этом для доминирующей платформы, чтобы сохранить свои позиции, постоянно приходится оптимизировать микро архитектуру. Цель этого – не допустить заметного отставания от других архитектурных платформ по скорости выполнения программ, поскольку замечено, что новая архитектурная платформа, существенно (в 2-3 или более раз) превосходящая сегодняшнего лидера, имеет хорошие перспективы на завоевание приверженности

сегодняшнего лидера, имеет хорошие перспективы на завоевание приверженности разработчиков программного обеспечения.

Использование виртуальных платформ, таких как Виртуальная Джава Машина с системой команд Java Byte Code [8] и .NET с системой команд MSIL [9], позволяет разработчикам, оттранслировав программу в коды соответствующей виртуальной машины, исполнять ее на любой аппаратной платформе. Для этого достаточно, чтобы на наиболее распространенных аппаратных платформах была программно реализована соответствующая виртуальная машина, что существенно проще, чем адаптация каждого программного приложения к каждой аппаратной платформе. Интересно, что для виртуальных платформ активно используются оптимизирующие динамические и статические компиляторы.

Однако методы оптимизирующей динамической двоичной трансляции все более активно начинают внедряться во вновь реализуемые архитектурные платформы, сохраняя для пользователя полную совместимость с доминирующей аппаратной платформой, но позволяя при этом кардинально изменить архитектуру микропроцессора и получить либо более высокую логическую скорость, либо меньшее энергопотребление. Это наблюдается на примерах архитектурной платформы EPIC, реализованной в семействе микропроцессоров IPF [5], для которых посредством динамической двоичной трансляции поддерживается совместимость с IA-32 на уровне приложений [4], в микропроцессорах фирмы Transmeta [6], в которых реализована полная совместимость для любых программ в кодах IA-32. Еще один пример – это архитектура широкой команды Эльбрус-3М, в которой мощно поддержаны программные методы оптимизаций и двоичной совместимости с IA-32 [1].

Результаты анализа использования компиляторных методов при реализации аппаратных и виртуальных платформ представлены в Табл. 1.

Табл. 1. Методы реализации аппаратных и виртуальных платформ

тип платформы	тип системы команд	представители	метод исполнения
аппаратная	CISC (сложная)	Intel Pentium3/4, AMD Athlon/Opteron	аппаратные двоичный транслятор и оптимизатор
аппаратная	RISC (упрощенная)	IBM Power4/5, Sun UltraSPARC	аппаратный оптимизатор
аппаратная	EPIC (с явным параллелизмом команд)	Itanium Processor Family	как предписано статическим компилятором с минимальной аппаратной оптимизацией
аппаратная	WI (EPIC)+C (с явным параллелизмом и поддержкой совместимости)	Transmeta Efficeon, Эльбрус-3М	как предписано статическим компилятором или программным динамическим двоичным транслятором
виртуальная	безадресная (стековая)	Java Virtual Machine, .NET CLI	программные интерпретатор, компилятор и оптимизатор

1.2. Базовая архитектура

Архитектура микропроцессора Эльбрус-3М характеризуется двумя принципиальными свойствами: она отличается предельно высокой архитектурной скоростью и обеспечивает полную совместимость с IA-32. Эти свойства поддерживаются технологией оптимизирующей языковой и двоичной компиляции, для которых, в свою очередь, предусмотрена специальная аппаратная поддержка. Рассмотрим особенности, поддерживающие высокую архитектурную скорость:

1. большой набор исполняющих устройств с возможностью явно управлять их запуском с использованием широкой команды (Табл. 2)
2. большой регистровый файл (Табл. 2), организованный в виде процедурных окон произвольной длины с автоматической откачкой/подкачкой окна при переполнении/исчерпании регистрового файла
3. наличие специальных операций и режимов исполнения, позволяющих преодолевать ограничивающие параллелизм зависимости:
 - режим спекулятивных (упреждающих) вычислений, которые нужны, чтобы уменьшить зависимости, связанные с условной передачей управления
 - исполнение операций под управлением предиката позволяет избавиться во многих случаях от операций передачи управления, что ведет к увеличению параллелизма и, как следствие, к росту архитектурной скорости
 - для преодоления статически неопределяемых зависимостей по данным предусмотрены операции спекулятивного (упреждающего) считывания данных в обход предполагаемой зависимости с последующим контролем корректности перестановки операций
4. специальные операции подготовки выполнения переходов, позволяющие распараллелить передачу управления по нескольким направлениям без потери тактов на выбор соответствующего пути
5. операции, поддерживающие лучшую адаптацию программы к иерархической памяти, которые представляют собой аппаратные средства синхронной и асинхронной предподкачки данных при регулярной обработке, а также средства управления размещением данных в кэшах различного уровня
6. аппаратная поддержка программной конвейеризации циклов, включающая в себя средства переименования регистров с использованием специальной вращающейся базы, а также счетчик цикла и специальные предикаты, управляющие циклом

Табл. 2. Архитектурные особенности микропроцессора Эльбрус-3М

тип операций	скалярные вычисления	конвейеризированные циклы
арифметические операции - с учетом упакованных типа SSE	до 10	до 10 до 16
обращения в память счит.+зап. - с учетом упакованных типа SSE	до 4+0 2+1 0+2	до 7+0 6+1 4+2 до 14+0 12+2 8+4
длинные литералы (2-байтные, 4-байтные, 8-байтные)	до 4(2б)+2(4б) 4(4б) 2(8б)	до 4(2б)+2(4б) 4(4б) 2(8б) (обычно не используются)
операций над предикатами - с учетом предиката от счетчика цикла	до 3	до 3 до 4
переходы (подготовка+исполнение)	1+1	1+1
продвижение счетчика цикла		1
используемых регистров - с учетом глобальных	до 64 до 96	до 192 до 256
пиковые возможности - с учетом упакованных типа SSE	до 16	до 23 до 33

Влияние перечисленных архитектурных свойств на производительность более подробно обсуждается при рассмотрении методов оптимизации и динамического анализа зависимостей в подразделах 2.2, 2.3 и 2.4.

Свойства базовой архитектуры, поддерживающие полную двоичную совместимость с архитектурой IA-32, разделяются на две составляющие: поддержка операционного базиса IA-32 и поддержка технологии оптимизирующей двоичной трансляции.

Основными компонентами поддержки операционного базиса IA-32 являются статистически наиболее значимые операции и архитектурные черты, аппаратная реализация которых существенно влияет на производительность:

1. целочисленные операции, включая упакованные (MMX/SSE2), и операции вычисления целочисленных флагов (поддержка последних не столь важна на высоких уровнях оптимизации, но необходима на промежуточных уровнях)
2. вещественные операции всех форматов (32, 64, 80), включая упакованные, а также плавающие сравнения, формирование накапливающих флагов исключений и поддержку плавающего стека (поддержка последнего важна на промежуточных уровнях оптимизации)
3. обращения в память, обеспечивающие совместимую семантику, включая доступ через сегментные регистры и выработку соответствующих исключений, возможность работы по не выровненным адресам, сохранение порядка обращений
4. наличие глобальных регистров, необходимое для сохранения x86-контекста между регионами оптимизации

Основными компонентами, поддерживающими технологию надежной, полностью скрытой от пользователя оптимизирующей двоичной трансляции, являются:

1. два виртуальных пространства, одно из которых содержит коды и данные IA-32, а другое используется для хранения откомпилированных кодов и самой системы двоичной трансляции, включая компиляторы нескольких уровней и систему динамической поддержки
2. механизм контрольной точки как средство обеспечения точного прерывания
3. средства контроля страниц, необходимые для обнаружения самомодифицирующихся кодов, а также для контроля обращений в специальные страницы ввода/вывода с особой семантикой для операций чтения/записи
4. кэш перекодирования адресов кода IA-32 в адреса кода микропроцессора Эльбрус-3М, необходимый для оптимизации передачи управления по динамически вычисляемым адресам между откомпилированными регионами
5. средства контроля памяти, поддерживающие перенос часто используемых данных из памяти на регистры процессора
6. все средства поддержки оптимизаций, предусмотренные для оптимизации языковых программ.

Первые три средства поддержки технологии двоичной трансляции необходимы в первую очередь для обеспечения корректной семантики исполнения оттранслированных двоичных кодов. Вопрос о том, насколько они важны с точки зрения производительности, требует отдельных исследований, часть из которых, относящаяся к поддержке вещественной арифметики, можно найти в работе [29]. Каждое из перечисленных средств дает прирост производительности от нескольких процентов до нескольких раз. Но поскольку на конечную производительность влияет много факторов, хорошо спроектированная архитектура – это компромисс между аппаратными затратами на то или иное свойство и эффектом от его включения в архитектуру.

2. Оптимизирующие компиляторы для базовой архитектуры

В рамках единого компиляторного проекта реализован целый комплекс компиляторов для базовой архитектуры. Во-первых, это статический оптимизирующий компилятор с языков C, C++ и Fortran-90 – ОКЭ. Он поддерживает также GNU C и GNU C++ (Рис. 1). Во-вторых, это динамический, полностью скрытый от пользователя двоичный транслятор, позволяющий исполнять любые коды IA-32, в частности, коды любых операционных систем. Наконец, это статический и динамический оптимизирующие двоичные трансляторы для приложений в кодах IA-32 (для всех оптимизирующих двоичных трансляторов используется сокращение ОДТЭ).

Сразу стоит сказать, что статический и динамический двоичные трансляторы приложений выполняют чисто технологические функции и не являются конечными продуктами, хотя динамиче-

ский двоичный транслятор может быть доработан до конечного продукта для исполнения IA-32 приложений под управлением OS Linux. На данном этапе эти трансляторы позволяют глубоко изучить различные аспекты оптимизации двоичных кодов и используются как великолепные средства отладки оптимизирующей двоичной трансляции.

Все трансляторы (языковые и двоичные) реализованы на базе единой инфраструктуры и используют единое платформу-зависимое промежуточное представление транслируемой программы (отдельного региона – для двоичных трансляторов). Конечно, семантика двоичных кодов обладает определенной спецификой, которая находит отражение в представлении, но общего между языковым и двоичным транслятором с точки зрения представления транслируемой программы значительно больше.

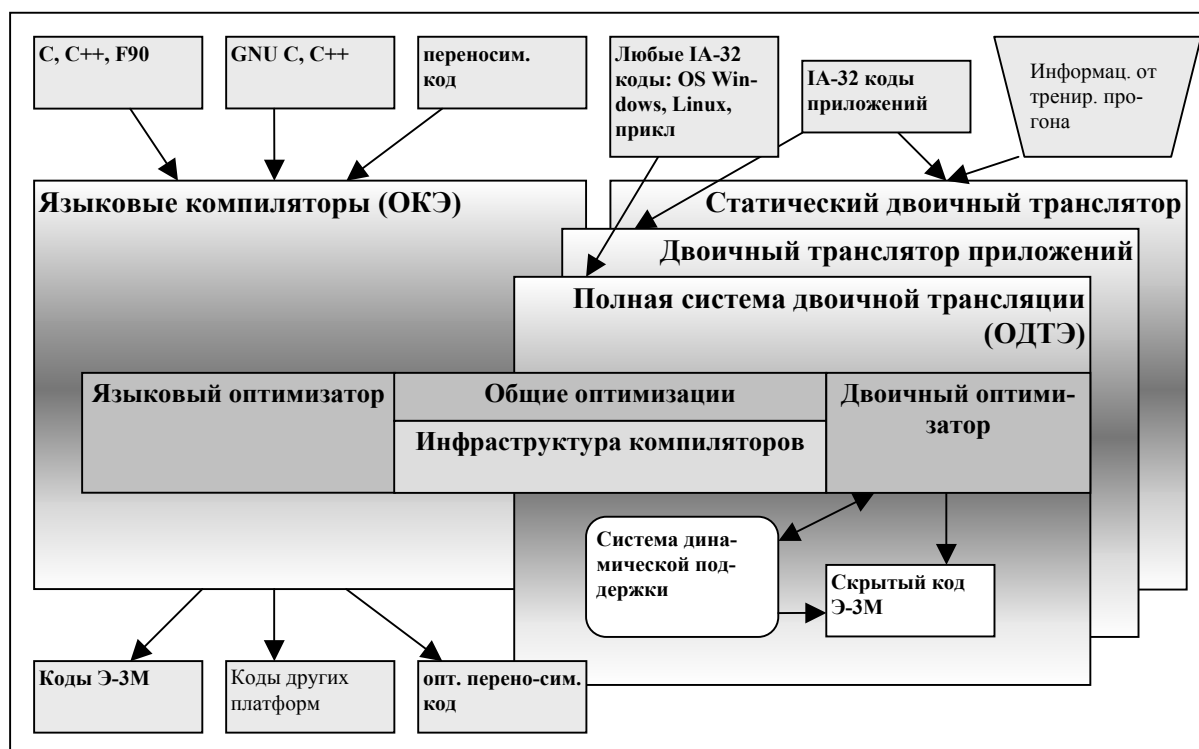


Рис. 1. Структура и состав компиляторных проектов

Алгоритмы оптимизаций для языковых и двоичных трансляторов реализованы независимо. В первую очередь это объясняется особенностями динамической трансляции. Поскольку время трансляции включается во время выполнения программы, необходимо использовать только быстрые алгоритмы оптимизации. Так, например, алгоритм распределения регистров на основе раскраски графа несовместимости [10] не может быть применен в динамическом трансляторе из-за квадратичной алгоритмической сложности. Технология оптимизации, при которой представление многократно обрабатывается отдельными алгоритмами анализа и оптимизаций, также недопустима в динамическом трансляторе. Приходится объединять алгоритмы вместе с тем, чтобы за один-два обхода представления выполнить все возможные оптимизации. Наконец, специфика двоичной семантики накладывает дополнительные ограничения на алгоритмы оптимизации.

В ограниченные рамки статьи невозможно вместить все методы оптимизации, используемые в компиляторах ОКЭ и ОДТЭ. Их описание можно найти, например, в [10, 11]. Поэтому рассматриваются только те из них, которые наиболее ярко отражают выбранный программно-аппаратный подход. Вследствие этого, за пределами статьи оказалось большинство универсальных методов

оптимизации, реализованных в ОКЭ и ОДТЭ, таких как вынос инвариантов из циклов, сбор общих подвыражений, понижение силы операций, удаление «мертвого» кода, множество эквивалентных преобразований управления, особенно для циклов, и другие.

2.1. Структурные изменения в компиляторах

Традиционный подход, используемый в языковых оптимизирующих компиляторах, это компиляция отдельного модуля с последующей сборкой модулей в исполняемый файл (Рис.2.а). Компилятор фактически представляет собой единую программу. На самом деле, его работа разделяется на фазы, две из которых необходимо выделить. Это языковая и оптимизирующая фазы компиляции. Языковая часть реализует полную семантику языка и отображает ее в простое промежуточное представление. Оптимизирующая фаза дополняет простое семантическое представление программы специальными структурами данных, используемыми в алгоритмах анализа и оптимизации, и получает на выходе семантически эквивалентное оптимизированное представление программы. Для архитектур с явным параллелизмом в оптимизирующую фазу включается также оптимальное планирование и распределение регистров, поскольку учет аппаратных ресурсов и их параллельных возможностей, с одной стороны, существенно повышает эффективность универсальных алгоритмов оптимизации а, с другой, существенно улучшает результаты планирования по сравнению с раздельным выполнением этих действий [16].

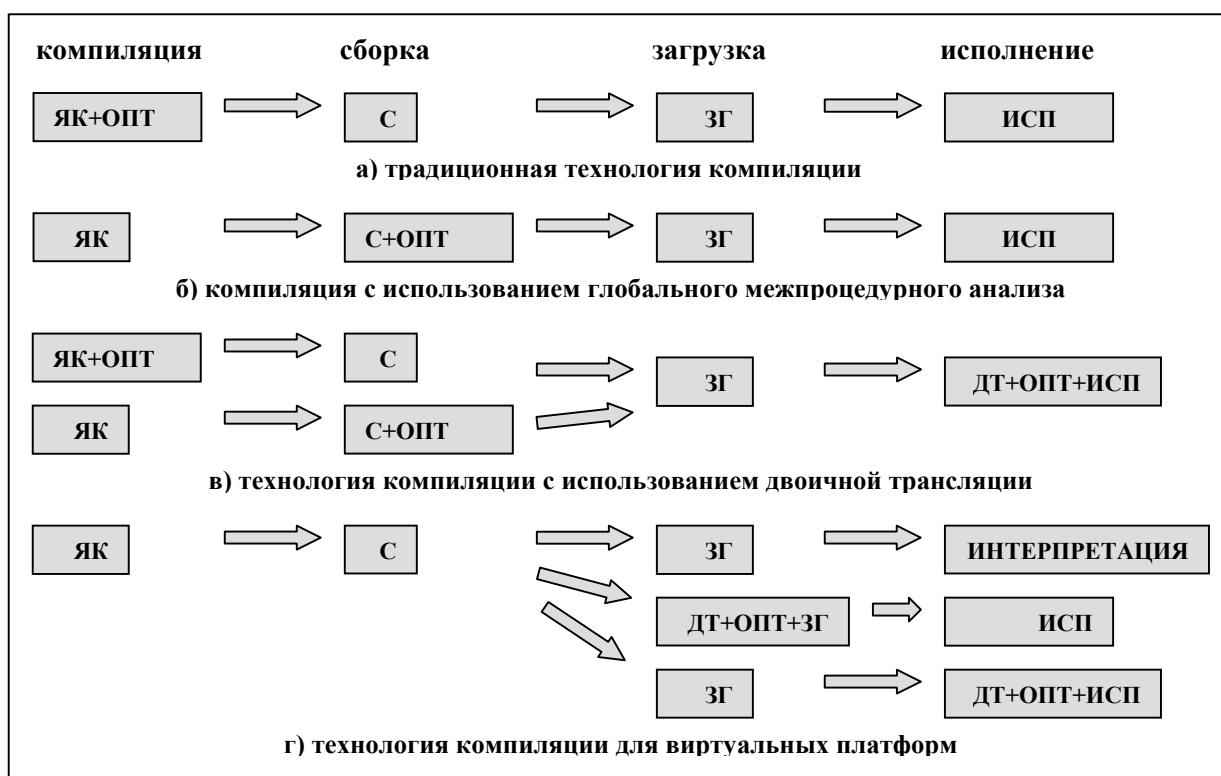


Рис. 2. Методы компиляции, оптимизации и исполнения программ

Методы глобального анализа и межмодульных оптимизаций позволяют существенно повысить производительность программ. Эффект от точного глобального анализа указателей на объекты, распространения констант и информации о выравнивании данных в сочетании с такими оптимизациями, как подстановка процедур в точку вызова и частичное клонирование процедур достигает 30% роста производительности. Для реализации глобального анализа и межмодульных оптимизаций

ций выполняется разрез компилятора. При трансляции модуля выполняется только языковая фаза, а оптимизирующая переносится на момент сборки. При этом программа сохраняется в виде промежуточного представления, представленного в очень компактном виде. В ОКЭ это представление в полтора раза более компактно, нежели эквивалентный код платформы IA-32, который считается минимальным относительно всех универсальных архитектурных платформ. В момент сборки программы сначала выполняются глобальный анализ и глобальная межмодульная оптимизация, а затем для каждого из модулей выполняется оптимизирующая фаза (Рис.2.б) и генерируется код целевой платформы, который сразу же поступает на сборку. Применительно к библиотекам возможна гибридная схема, когда для модуля вместе с кодом целевой платформы формируется компактное промежуточное представление. Этот подход позволяет использовать библиотечные модули непосредственно с целью сборки на уровне кода, а также позволяет при необходимости включать их в глобальную межпроцедурную оптимизацию.

Для динамического двоичного транслятора процесс оптимизации распадается на две фазы: первоначально программа оптимизируется под одну платформу, а затем в процессе исполнения – под другую (Рис. 2.в). С одной стороны, при оптимизации двоичного кода, как уже отмечалось выше, значительная часть статической информации о программе не попадает в двоичный оптимизирующий транслятор, поскольку она не сохраняется в исполняемом коде, что снижает оптимизирующие возможности. Но, с другой стороны, двоичный код, поступающий в динамический оптимизатор, уже хорошо оптимизирован в статическом компиляторе. Динамическая же оптимизация обладает значительными адаптивными возможностями, так что совокупный эффект от такой двухфазной оптимизационной схемы может скомпенсировать потери от недостатка статической информации о программе. Динамическая двоичная трансляция использовалась как метод оптимизации программ [12, 13], когда коды некоторой платформы дополнительно подвергались динамической оптимизации, и этот подход показал высокую эффективность их выполнения, сопоставимую с самыми высокими уровнями оптимизации для данной платформы. Другой подход – это когда коды одной платформы попадают в динамический оптимизатор другой платформы для обеспечения полной программной совместимости. Этот подход лучше всего реализуется на архитектурах с явным параллелизмом команд и с аппаратной поддержкой процесса двоичной трансляции. Дополнительным преимуществом динамической трансляции является возможность легко преодолевать межпроцедурные и межмодульные границы, выполняя тем самым самый мощный класс межпроцедурных оптимизаций. В ОДТЭ благодаря адаптивной оптимизации и достаточно хорошей аппаратной поддержке удается выполнять программы, исполняющиеся существенно быстрее, нежели на исходной платформе при той же тактовой частоте.

Для компиляторов с языков в виртуальные архитектуры оптимизатор может выполняться как в момент загрузки, так и в момент исполнения программы (Рис. 2.г). При этом, в отличие от случая двоичной трансляции, в момент работы динамического оптимизатора доступна статическая информация об объектах программы, поскольку она сохраняется в двоичном коде виртуальных платформ, что потенциально повышает оптимизирующую мощь данного подхода.

2.2. Анализ зависимостей

Методы анализа программы применяются для улучшения условий применимости тех или иных оптимизаций или других видов анализа. Так, например, распространение констант, определяющих границы цикловых индексов, существенно уощняет возможности анализа зависимостей в циклах.

В компиляторах для базовой архитектуры используется большой набор методов анализа зависимостей. Они разделяются на статические и динамические. Статические методы действуют при компиляции программы, а динамические – при ее исполнении.

Одним из важнейших методов устранения ложных зависимостей является распределение всех обращений за данными по объектам. При этом считается, что обращения к разным объектам не могут приводить к конфликтам и, как следствие, считаются независимыми (Рис. 3.а). Этот метод хорошо известен и используется во многих современных оптимизирующих компиляторах. Он хо-

рошо работает, когда в программе встречаются обращения по именам к глобальным или локальным переменным. Но если ссылка на объект приходит через параметр или через глобальный указатель (pointer), а именно это, как правило, наблюдается в реальных программах, необходим дополнительный вид анализа – вычисление возможных имен статических объектов, которые могут содержаться в указателе, так называемый pointer tracking. Такой анализ программ является чрезвычайно сложным по времени, а его точность зависит от глубины. В компиляторе для базовой архитектуры введено несколько уровней глубины анализа, различающихся учетом управления и контекста. Реализованные алгоритмы позволяют выполнять самый углубленный уровень анализа за разумное время и с разумным размером памяти, используемой для объектов, в то время как менее глубокий анализ выполняется достаточно быстро и не требует много памяти.

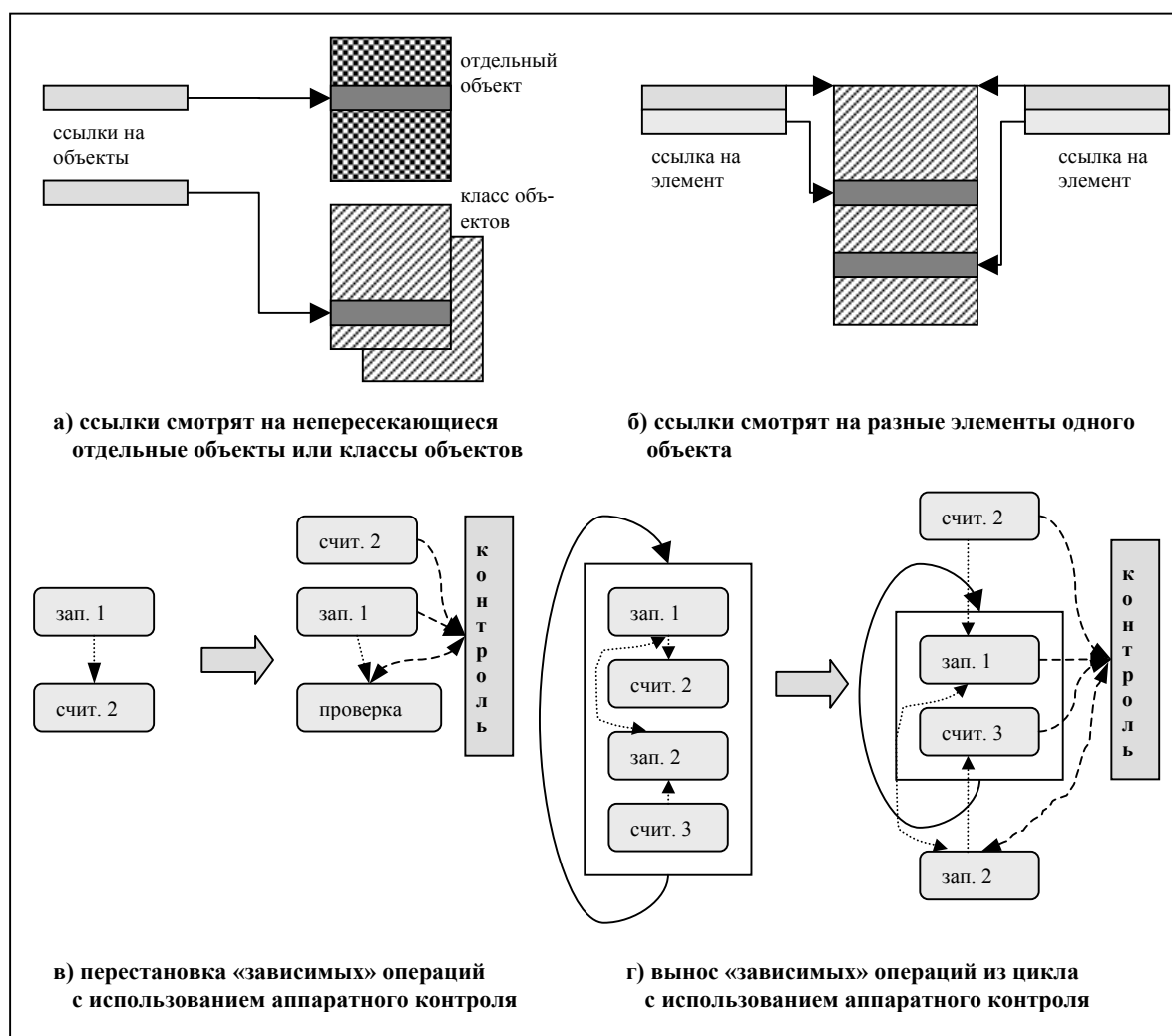


Рис. 3. Зависимости и методы их устранения

Следующим видом анализа зависимостей является индексный анализ. Его задачей является обнаружение независимости между двумя обращениями к одному и тому же объекту (многомерному массиву) по динамически вычисляемым индексам (Рис. 3.б). Этот вид анализа полезен для выявления всех видов параллелизма. Он особенно в случае анализа зависимостей в циклах, поскольку при обнаружении отсутствия зависимостей между итерациями, можно выполнять любое распараллеливание цикла, а при наличии ограниченных зависимостей сохраняется возможность выполнять программную конвейеризацию цикла. В нашем компиляторе реализовано несколько алгоритмов

такого анализа. При этом было проведено углубленное исследование эффективности различных методов и показано, что использование симплекс-метода применительно к рациональным числам позволяет наиболее быстро находить решение систем неравенств, по которым определяется наличие или отсутствие зависимостей. Эти результаты [14], превосходят многие из ранее опубликованных в этой области. Конечно, полный индексный анализ, в основе которого лежит решение систем Диофантовых уравнений и систем целочисленных неравенств, делинеаризация индексных выражений и многое другое, возможны только в статическом языковом компиляторе. Тем не менее, упрощенные варианты индексного анализа, в частности, анализ зависимостей между двумя обращениями в память по одному и тому же динамическому адресу с различными константными смещениями, применяются также в динамическом компиляторе.

Только статических методов анализа зависимостей оказывается недостаточно, поэтому часть анализа зависимостей выполняется динамически. Хотя эти действия, включаемые в код оптимизируемой программы, являются избыточными, в случае успешного анализа позволяют распараллелить программу и, как следствие, выполнить ее быстрее. В ОКЭ и в ОДТЭ используется несколько разновидностей подобного анализа.

Во-первых, это упрощенный вариант индексного анализа, результат которого выявляет, что два обращения в память не пересекаются либо потому, что адреса пробегают по непересекающимся отрезкам памяти, либо они пробегают по разным элементам одного и того же отрезка, поскольку изменяются по непересекающимся смещениям (простейший случай отсутствия решения Диофантового уравнения). Эта разновидность анализа, применяемая к циклу целиком, позволяет в случае успеха передать управление на параллельную версию цикла и значительно ускорить его выполнение. Наши исследования показали, что для задач с плавающей арифметикой данные методы динамического анализа зависимостей позволяют в несколько раз ускорить выполнение цикла. При этом сами проверки составляют лишь незначительную часть времени. Специальные исследования, проведенные на пакете тестов SPECfp95, выявили, что динамические проверки замедляют исполнение программы не более чем на 5% по сравнению со статическим анализом.

Во-вторых, применяется более локальный анализ зависимостей, осуществляемый методом сравнения между собой зависимых обращений в память. В этом случае ОКЭ либо вставляет в код программы явные проверки, что два обращения в память не имеют пересекающихся частей, либо использует аппаратный механизм контроля перестановок обращений в память (Рис. 3.в), описанный в разделе 1.2. Сравнивая между собой два данных способа, следует отметить, что метод динамического сравнения адресов требует включения в код большего числа проверок, поскольку они необходимы для каждой пары переставляемых обращений в память, в то время как перестановка на базе аппаратного контроля требует проверки только для переставленных операций считывания. С другой стороны, перестановка на основе аппаратного контроля в отдельных случаях требует дополнительного (компенсирующего) кода и большего числа регистров, используемых для восстановления в случае некорректной перестановки. Специальная эвристика используется для выбора конкретного способа, что дает значительный эффект в повышении скорости выполнения программ, устраняя с критических путей ложные, реально не существующие зависимости и снимая ограничения с возможности программной конвейеризации циклов.

Наконец, еще один вид устранения зависимостей с использованием аппаратной поддержки (Рис. 3.г), позволяющий не только избавиться от ложных зависимостей, но и сократить количество обращений в память, более рассмотрен в разделе 2.4.

2.3. Оптимизации для платформ с явным параллелизмом

При оптимизации программы применительно к платформам с явным параллелизмом команд она разбивается на отдельные регионы, к каждому из которых применяются различные стратегии и используются различные средства аппаратной поддержки. Поскольку параллелизм задается явно, целью оптимизации является наиболее эффективное использование параллельных аппаратных

ресурсов. Чтобы использовать их в полной мере, необходимо иметь как можно больше команд в оптимизируемом регионе. Поэтому в регионы для оптимизации включаются совокупности процедур, частично или полностью подставляемые в точку вызова.

С точки зрения стратегий распараллеливания все регионы можно разделить на три категории: *ациклические*, *конвейеризируемые циклы* и *смешанные* (регионы со сложным управлением). Следует отметить, что наибольшего эффекта от оптимизации удастся добиться с использованием адаптивных методов, описываемых в 2.5.

Ациклический регион представляет собой совокупность базовых блоков, связанных друг с другом по управлению (Рис. 4.а). Как правило, в него включаются базовые блоки, обладающие сопоставимой частотой выполнения. Для архитектур, поддерживающих явно предикатные и спекулятивные вычисления, наиболее изученным регионом является гиперблок, который представляет собой совокупность базовых блоков с одним входом и многими выходами [15]. Оптимизация такого региона состоит из предварительного преобразования его в предикатно-спекулятивное представление [17], целью которого является устранение всех внутренних зависимостей по управлению и перевод их в предикатную форму. При этом переходы из гиперблока наружу сохраняются. Далее над полученным представлением выполняются оптимизации критических путей [18], после чего оно поступает на планирование.

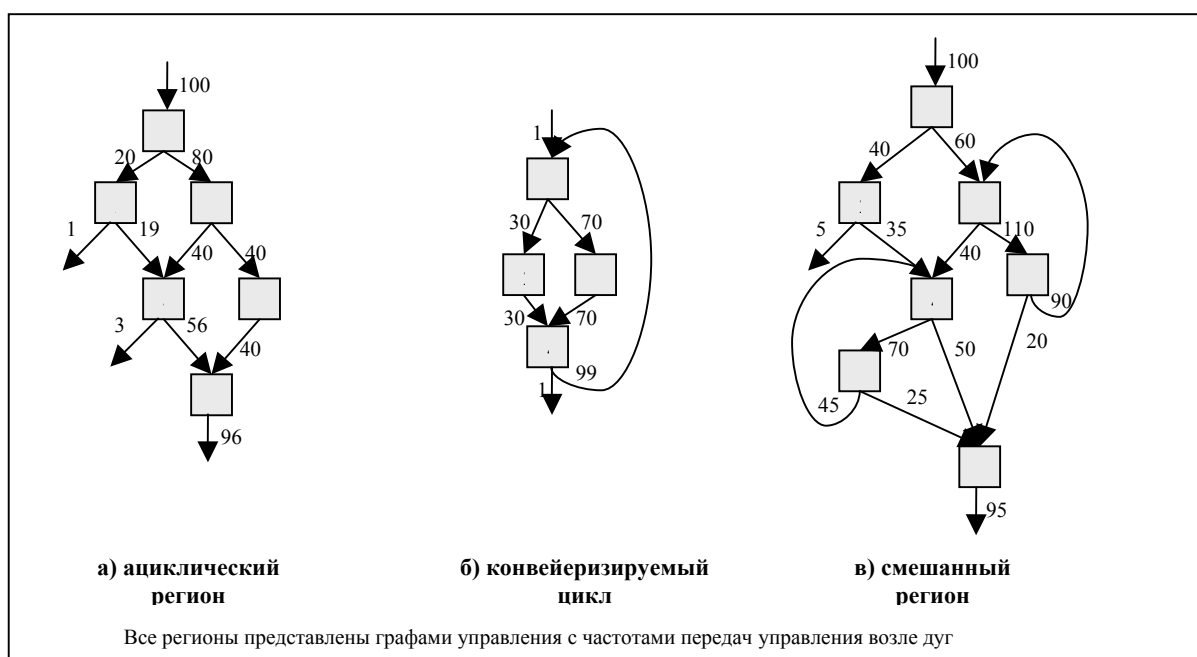


Рис. 4. Важнейшие категории регионов оптимизации

Для планирования ациклических регионов используется усовершенствованный алгоритм планирования списков. При наличии достаточно большого набора параллельных исполняющих устройств в широкой команде Эльбруса-3М (Табл. 2), с одной стороны, и, как правило, ограниченного параллелизма гиперблока – с другой, алгоритм планирования списков позволяет получить очень хорошее отображение абстрактного параллельного представления программы на конкретные аппаратные ресурсы.

Конвейеризируемый цикл – это регион, позволяющий совмещать выполнение нескольких последовательных итераций цикла. Как правило, этот метод оптимизации применяется к самым внутренним циклам, которые повторяются много раз (Рис. 4.б). В базовой архитектуре он имеет

мощную аппаратную поддержку. Практически все аппаратные средства поддержки оптимизаций задействуются при этой оптимизации. Сначала тело цикла преобразуется в один гиперблок, причем все выходы также объединяются в один. Для этого активно используются предикатные и спекулятивные операции, а также логические операции над предикатами. Для устранения блокировок при обращении в память используется аппаратная техника предварительной подкачки данных в специальный буфер. Использование этого механизма дает дополнительное преимущество, поскольку при этом исполняющие устройства освобождаются от операций обращения в память и могут быть целиком использованы только для арифметических операций. Наконец, аппаратная поддержка базированных вращающихся регистров позволяет полностью избавиться от ложных зависимостей между операциями и исключить избыточные операции копирования, которые обычно необходимы, чтобы избежать указанной проблемы.

При конвейеризации цикла преследуется одна из двух целей: либо добиться предельной загрузки имеющихся аппаратных ресурсов, либо достичь критического пути рекурсивной цепочки зависимостей. Комплекс оптимизаций, решающих обе эти проблемы, подробно описан в [19]. В этой работе показано, что аппаратная поддержка циклов методом их конвейеризации дает существенный (в 2,5-3 раза) прирост производительности на задачах с вещественной арифметикой по сравнению с чисто программными методами. И даже на целочисленных задачах конвейеризация циклов с аппаратной поддержкой дает небольшое, порядка 7%, улучшение по сравнению с чисто программными методами конвейеризации. Но это объясняется тем, что целочисленные задачи содержат очень мало самых внутренних циклов, повторяющихся много раз. Для них более характерны смешанные регионы, о которых говорится далее.

Для конвейеризируемых циклов используются специальные методы планирования. В отличие от планирования ациклического региона, для которого характерно последовательное и однократное наполнение широких команд операциями, планирование широкой команды конвейеризируемого цикла выполняется несколько раз. Это объясняется тем, что в одну широкую команду могут попадать операции с разных стадий программного конвейера (с разных итераций) цикла. При этом усовершенствованный алгоритм планирования списков готовых операций оказывается успешным, но на его работу накладываются дополнительные ограничения, вызванные необходимостью решать более сложные проблемы конфликтов между операциями при назначении для них нужных исполняющих устройств.

Смешанный регион или регион со сложным управлением представляет собой совокупность ациклических и цикловых регионов, в которых работа внутри циклов не столь продолжительна, как для конвейеризируемых циклов (Рис.4.в). В целочисленных приложениях зачастую встречаются циклы, среднее время выполнения которых не превышает двух раз. Попытка оптимизировать такие циклы методом их конвейеризации не только не ускоряет, а зачастую замедляет выполнение программы, поскольку длина программного конвейера может в несколько раз превысить число повторений цикла. Для таких циклов допустима только конвейеризация с полной откруткой пролога [19], но даже она не всегда приводит к успеху, поскольку основная цель конвейеризации – предельная загрузка аппаратных ресурсов даже ценой удлинения времени выполнения одной итерации – неприемлема для циклов с малым числом повторений. Поэтому к этим регионам применяются методы глобального планирования, соединяющие воедино оптимизацию и планирование команд.

Основная идея глобального планирования состоит в том, что при распараллеливании вычислений сразу учитывается его влияние на производительность. Наибольшие проблемы при распараллеливании регионов со сложным управлением вызывает планирование точек схождения управления внутри региона. Большинство таких точек приходится на входы в циклы и выходы из них. Семантически эквивалентное преобразование программы, при котором операции переносятся через данные точки схождения лежит в основе глобального планирования. При этом учитываются влияние таких преобразований на критические пути выполнения различных частей региона, включая

циклы и ациклические участки [20]. В конечном итоге циклы оказываются частично конвейеризированными, и при этом в них самих и в ациклических участках активно используются предикатные и спекулятивные операции. Предварительные результаты показывают, что методы глобального планирования при той аппаратной поддержке, которая заложена в базовой архитектуре, позволяют повысить эффективность исполнения регионов со смешанным управлением на 20-50% по сравнению с независимым планированием циклов и ациклических регионов.

Несмотря на множество работ, выполнявшихся в течение почти 20 лет в направлении поиска универсальных методов оптимизации для архитектур с явным параллелизмом, этот процесс до сих пор не завершен. Работа активно продолжается как для IPF, так и для архитектуры Эльбрус-3М. При этом исследования и анализ ведутся по трем направлениям:

1. поиск новых оптимизаций с учетом имеющейся аппаратной поддержки
2. контроль качества уже реализованных оптимизаций
3. поиск новых методов оптимизаций на базе усовершенствованной аппаратной поддержки для них.

Результаты таких исследований представлены в [21]. Эффективность этой работы трудно переоценить. Так только за один год за счет предложений, полученных от этой аналитической группы, удалось улучшить производительность компилятора на целочисленных приложениях более, чем в 2 раза, а на вещественных – почти в 6 раз. Конечно, эти показатели достигнуты не только за счет специальных оптимизаций, связанных с особенностями базовой архитектуры. Они включают в себя предложения по улучшению анализа, по применению различных эквивалентных преобразований управления, по усовершенствованию планирования и распределения различных типов регистров, по оптимизации обращений в память и много другое. Но конечная цель всех этих предложений – улучшение качества оптимизаций для платформы с явным параллелизмом команд.

2.4. Оптимизация обращений в память

Анализ трасс выполнения современных программ показывает, что значительные потери производительности происходят из-за обращений в память (Рис. 5.а). Поскольку разрыв между временем выполнения операций в процессоре и операциями обращения в память продолжает увеличиваться, для сокращения потерь в аппаратуру встраивается все более глубокая иерархия скрытых памятей – кэшей. Тем не менее, чисто аппаратный подход не решает проблему, и оптимизация обращений в память в компиляторах представляет собой важный резерв повышения скорости выполнения программ.

Можно выделить три основных стратегии оптимизации обращений в память:

- сокращение числа обращений за счет переноса данных из памяти на регистры или объединения нескольких обращений в одно
- заблаговременная подкачка данных в специальные буфера и близкие уровни кэшей или предсказания хранящихся в них значений
- оптимальное размещение данных в памяти и на различных уровнях кэшей, сокращающее потери от фрагментации или травления и тем самым улучшающее использование кэшей

Сокращение числа обращений в память – это хорошо известная оптимизация, устраняющая повторные обращения в пределах региона оптимизации [10]. Суть ее состоит в том, что на основе анализа компилятор обнаруживает повторные обращения в память и удаляет их из кода, оставляя минимально необходимое число обращений, сохраняя результаты в регистрах (Рис. 5.б). Эта оптимизация особенно эффективна в циклах, поскольку устраняет не только лишние операции, но и возможные конфликты в кэше. Применение этой оптимизации ограничивается возможными зависимостями, зачастую ложными, когда обращение в память выполняется через указатель (pointer). При трансляции программы с языка такие проблемы возникают реже, поскольку статический компилятор может распознать много независимых объектов. Но для двоичного транслятора очень часто между двумя обращениями к одной и той же переменной в памяти встречаются обращения к

другим данным, что резко снижает возможности анализа и, как следствие, подобной оптимизации. В базовой архитектуре предусмотрено специальное средство, позволяющее выполнять такой перенос данных, сохраняя при этом контроль над возможными конфликтами при обращении по другим адресам (Рис. 3.г). Техника оптимизации с использованием данной аппаратной поддержки широко используется в ОДТЭ и дает существенный (10-20%) прирост производительности на горячих регионах.



Рис. 5. Методы сокращения потерь при обращении в память

Сокращение числа обращений в память за счет объединения нескольких обращений в одно используется при обработке смежно расположенных данных, совокупный формат которых не превышает максимально допустимый формат аппаратной команды. Такая оптимизация может применяться даже при обработке 32-разрядных значений, но наибольшего эффекта удастся добиться при обработке данных меньшего формата (однобайтных и двухбайтных). В сочетании с имеющимися в базовой архитектуре, как, впрочем, и во многих других архитектурах, операциями над упакованными значениями (операции типа MMX/SSE) этот метод оптимизации позволяет существенно поднять производительность целого класса задач и даже находит успешное применение в универсальных приложениях [24].

Предварительная подкачка данных в буферные памяти или в кэш является эффективным средством оптимизации обращений в память. В большинстве современных микропроцессорных архитектур предусмотрены специальные операции, реализующие такие функции. Как правило, предварительная подкачка выполняется в кэш. В некоторых архитектурах, например, UltraSPARCIII [7] кэш специально структурируется, то есть для предварительно подкачиваемых данных предусматривается специальный кэш. Такой подход объясняется стремлением избежать выдавливания полезных данных из обычного кэша, что происходит при прокачке через него

лезных данных из обычного кэша, что происходит при прокатке через него больших массивов данных. Тем не менее, даже такое решение не позволяет полностью избавиться от проблемы приостановки вычислений из-за неизвестного времени доступа в память, поскольку предварительная подкачка планируется статически в компиляторе (Рис. 5.в).

В базовой архитектуре наряду с предварительной подкачкой данных в кэш предусмотрены средства предварительной подкачки элементов массивов в цикле. Эти средства содержат автономные регистры, с помощью которых вычисляются регулярно изменяющиеся адреса, а также специальную буферную память, предназначенную для хранения заблаговременно считанных элементов массивов. Благодаря гибкой организации, механизм буфера предварительной подкачки обеспечивает асинхронную подкачку данных различных форматов, поддерживая наполнение соответствующих областей предварительно подкачиваемых данных. Подобная организация полностью снимает проблему приостановки вычислений ввиду неизвестности времени доступа в память, поскольку асинхронная подкачка фактически реализует адаптацию к произвольному времени доступа. Таким образом, если все данные окажутся в кэше, время предварительной подкачки будет коротким, а если данные полностью или частично будут поступать из памяти, время предварительной подкачки будет не больше времени одного максимального доступа в память на весь цикл обработки (Рис. 5.г). Из буфера предварительной подкачки данные передаются в регистры процессора специальными командами. При этом команды выполняются устройством предварительной подкачки и не требуют использования арифметико-логических устройств, что дает дополнительные резервы производительности.

Использование средств асинхронной предварительной подкачки данных в буфер и предварительной подкачки данных в кэш является одним из важных резервов повышения производительности в ОКЭ. Обычно, чтобы обеспечить бесперебойность вычислений, данные для цикла с наибольшим уровнем вложенности подкачиваются с использованием буфера предварительной подкачки, а для гнезда используется предварительная подкачка данных в кэш для нескольких начальных итераций следующего поколения этого цикла. Такой подход дает существенный прирост производительности, особенно на задачах с плавающей арифметикой, в которых доминируют циклы [19].

Сокращение обращений в память за счет предсказания сохраняемого в ней значения является еще одной разновидностью заблаговременной подкачки данных. Эта техника опирается на довольно дорогостоящий метод профилирования программы с целью выявления обращений с хорошо предсказуемыми значениями и требует получения создания двух версий кода, поэтому она пока не нашла достаточно широкого применения в компиляторах.

Еще одной стратегией уменьшения потерь от обращений в память является оптимальное размещение данных. Линейная память, используемая для хранения данных, а также ограниченные размеры, строковая организация и ограниченная ассоциативность кэшей создают проблемы, известные как фрагментация, и приводят к возникновению дополнительных конфликтов. Разработчики аппаратуры постоянно заняты поиском новых методов организации кэшей [22]. Наряду с этим продолжается активный поиск программных методов сокращения потерь от неудачного размещения данных. Базовые идеи описаны в статье [23]. Наряду с группировкой данных по частоте их использования (рис. 5.д) довольно перспективным представляется перераспределение данных (создание копий) перед регионами интенсивной обработки. Большинство из упомянутых методов оптимизации реализовано в ОКЭ.

Наконец, в таких архитектурах как Itanium и Эльбрус-3М предусмотрены программные средства управления размещением данных на нужном уровне кэша. Смысл этой оптимизации заключается в стремлении избежать ненужного выдавливания данных из самых быстрых кэшей и тем самым повысить вероятность сохранения в них полезной информации. В ОКЭ такие оптимизации применяются к циклам, в которых выполняется обращение за данными, не препятствующее конвейеризации (такие обращения можно не помещать в кэш первого уровня). При этом выполнение одной

итерации немного замедляется, но за счет сохранения в кэше первого уровня более важных данных общее время выполнения цикла сокращается.

2.5. Адаптивная оптимизация

Для архитектур с явным параллелизмом адаптивные методы оптимизации дают существенный прирост производительности. К адаптивным относятся методы, использующие информацию о поведении программы. Для обычных статических компиляторов эти методы сводятся к профилированию программ и отдельных данных с тем, чтобы решить проблемы непредсказуемого поведения в точках ветвления управления, а также профилированию данных, результаты которого позволяют существенно упрощать вычисления за счет того, что при компиляции известны наиболее вероятные значения некоторых часто используемых переменных.

Однако динамические методы профилирования требуют выполнения так называемых «тренировочных» прогонов программы на специальных «тренировочных» данных. Это действие не всегда является технологически удобным для пользователя. Кроме того, оно обладает еще одним недостатком – тренировочный запуск может отличаться от реального. В данном случае эвристики, оптимизирующие программу на основе тренировочного запуска, могут сформировать код, который на реальных данных будет работать плохо. Тем не менее, этот метод является очень эффективным для подавляющего большинства программ, причем его использование позволяет повышать производительность откомпилированных программ не только для архитектур с явным параллелизмом, но и для суперскалярных архитектур (хотя для последних эффект от этого класса оптимизаций значительно меньше).

Основное, что дает метод адаптивной оптимизации – это возможность сгруппировать команды в соответствии с частотой их использования и уменьшить потери от неверно предсказанных направлений перехода. Для архитектур с явным параллелизмом имеется дополнительная возможность – использование предикатных и спекулятивных операций. Эти преимущества позволяют за счет дополнительных вычислительных ресурсов и выполнения одновременно нескольких альтернатив вообще избавиться от многих переходов и, тем самым, устранить потери от неверно предсказанных или своевременно не подготовленных переходов. Возможности частичных предикатных вычислений имеются и во многих суперскалярных архитектурах – это операции пересылки, выполняющиеся в зависимости от значения регистра. Специальные исследования [25, 26] показывают, что использование полных предикатов, т.е. возможность выполнить любую операцию под управлением предиката, в сочетании со спекулятивным режимом выполнения повышают производительность целочисленных программ на 20-25% по сравнению с архитектурами, не использующими этих возможностей, в то время как использование только частичных предикатов дает прирост производительности не более, чем на 10%.

Однако наличие предикатных и спекулятивных операций не может быть эффективно использовано без применения адаптивных методов оптимизации. Это особенно важно для целочисленных приложений. Характерной особенностью таких программ является сильное ветвление управления и очень короткие базовые блоки. Фактически на каждые 3-4 операции приходится одна команда передачи управления, причем одна из операций, как правило, является операцией сравнения. Попытка выполнять программу по всем возможным путям управления приводит к катастрофическим результатам, поскольку для таких задач, как, например, компиляторы, это приводит к тому, что свыше 90% вычислений выполняются напрасно, а время выполнения не только не сокращается, но и существенно возрастает. Поэтому так важно иметь информацию о вероятных путях исполнения и использовать ее для выделения наиболее интенсивных вычислительных регионов.

Многочисленные исследования показали хорошую предсказуемость поведения программ [7]. Однако чисто динамические методы профилирования обладают еще одним недостатком кроме не совсем достоверного предсказания, а именно: они не дают никакой информации о поведении программы на тех участках, которые не исполнялись во время тренировочных прогонов. Этой про-

блемы можно избежать, если использовать статические методы предсказания переходов [27], хотя последние значительно менее достоверны (вероятность неверного предсказания перехода составляет порядка 25%, в то время как для динамических предсказателей она менее 7%). В ОКЭ была применена гибридная схема, при которой динамическое предсказание используется для усовершенствования эвристик статического предсказания переходов [28]. Этот подход оказался очень эффективным и позволил в 2 раза улучшить характеристики статического предсказателя переходов. Таким образом, сочетание статических и динамических методов предсказания переходов (Рис.6.а) обеспечивает компилятор эффективным инструментом для проведения адаптивных методов оптимизации.

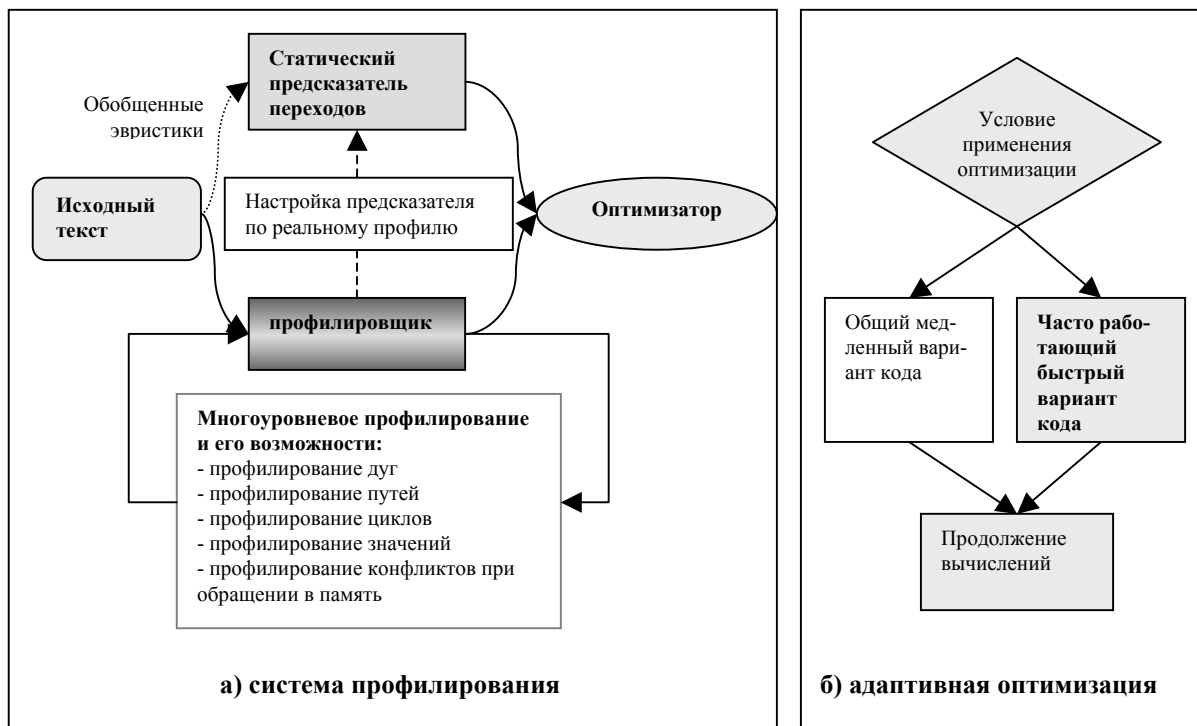


Рис. 6. Система профилирования и адаптивная оптимизация

Для динамических компиляторов применяется профилирование во время исполнения программы. С этой целью используются как чисто программные, так и программно-аппаратные методы сбора информации о поведении программы. Важность адаптивных методов оптимизации столь высока, что в архитектуры встраивается все больше средств, облегчающих сбор информации о поведении программы. Наряду с информацией о переходах эти средства позволяют собирать информацию о промахах при обращении к кэшам и других динамических событиях, снижающих эффективность выполнения программы. Использование этой информации в динамическом компиляторе позволяет наиболее оптимально выделять в программе регионы, требующие агрессивных методов оптимизации.

Ярким примером адаптивной оптимизации может служить метод оптимизации региона, содержащего вещественные вычисления, представленные в кодах x86 [29]. Идея оптимизации состоит в том, что на границах региона *динамически* проверяются некоторые условия, которые верны в подавляющем большинстве случаев (Рис.6.б). Если условия выполнены, оптимизатор получает дополнительную информацию о свойствах программы, на основе которой ему удастся выполнить гораздо более мощные оптимизации. В противном случае (крайне редко) выполняется не оптимизированный код, который не оказывает никакого влияния на итоговую производительность программы. Интересно, что подобный подход может быть применен и с

граммы. Интересно, что подобный подход может быть применен и с успехом применяется в языковых компиляторах. Фактически, он дополняет статический анализ программы. Известно, что далеко не все свойства программы, необходимые для применения тех или иных оптимизаций, могут быть вычислены при компиляции. Динамический анализ программы позволяет преодолеть эту проблему. Конечно, в основе такого подхода лежит знание о поведении программы или об отдельных типичных семантических свойствах. Все это присуще динамическим оптимизаторам, а статические стараются получить подобную информацию о программе, используя технику профилирования в процессе предварительного тренировочного исполнения оптимизируемой программы.

2.6. Быстрые алгоритмы оптимизации

Для достижения высокой производительности компилируемых программ современные оптимизирующие компиляторы используют множество различных алгоритмов оптимизации. Так, например, в ОКЭ используется свыше 150 различных алгоритмов. Многие из этих алгоритмов требуют для своей работы полной обработки промежуточного представления программы. И хотя большинство из них обладают линейной сложностью по отношению к обрабатываемым структурам данных, чисто последовательное их применение существенно замедляет время компиляции. В ОКЭ разработано несколько подходов к оптимизации алгоритмов. Одним из наиболее эффективных можно считать метод, при котором оптимизации объединяются в группы и применяются одна за другой при однократном обходе промежуточного представления. Как показано в [30], такой подход позволяет значительно сократить время компиляции. Еще более агрессивно этот метод применяется в ОДТЭ. Там большинство потоковых оптимизаций собрано всего только в два обхода промежуточного представления оптимизируемого региона: один - в прямом, а другой - в обратном направлении по графу управления и графу зависимостей. Это позволило в несколько раз сократить время работы оптимизатора. Конечно, такой подход имеет свои сложности, поскольку отлаживать сгруппированные, сильно влияющие друг на друга преобразования становится значительно труднее. Для их отладки приходится встраивать в компиляторы специальные верификационные средства, а также разрабатывать направленные тесты для достижения промышленного уровня надежности.

Некоторые алгоритмы обладают нелинейной сложностью. К ним относятся наиболее важные с точки зрения конечной производительности алгоритмы, такие, как, например, глобальное распределение регистров или оптимизация критических путей. Одним из способов сокращения времени их исполнения является сокращение размеров регионов, к которым они применяются. Но для архитектур с явным параллелизмом такой подход может приводить к недоиспользованию вычислительных возможностей, поэтому усовершенствование алгоритмов является наиболее предпочтительным направлением при решении указанных проблем. В ОДТЭ разработан оптимальный алгоритм устранения зависимостей [31], более чем на порядок превышающий по скорости известные до настоящего времени алгоритмы. Достоинством алгоритма является и то, что с его помощью можно оптимизировать критические пути в ациклических регионах на предикатном представлении программы. Важность этих результатов чрезвычайно высока, особенно для архитектур с явным параллелизмом команд.

Еще одним методом сокращения времени компиляции является адаптивная оптимизация. Использование информации о поведении программы позволяет применять наиболее агрессивные и, как следствие, наиболее вычислительно сложные алгоритмы, преимущественно к наиболее «горячим» регионам. Многократно проведенные исследования и наши собственные результаты показывают, что для большинства целочисленных задач более 90% времени приходится на исполнение менее 10% статического кода программы. Поскольку места сосредоточения этого кода достаточно надежно выявляются с помощью профилирования, можно наиболее мощно оптимизировать только эти участки программы, применяя к остальным менее сложные алгоритмы. Такой подход также позволяет существенно ускорить компиляцию.

3. Надежность оптимизирующих компиляторов

Вопросы надежности компиляторов занимают исследователей практически с момента появления первых компиляторов. Причина проста – компилятор может внести искажения в правильную программу, поскольку он является посредником между автором программы и компьютером. Единственно надежным способом получения безошибочно работающего компилятора является формальное доказательство корректности его работы. Но, к сожалению, ни для ранних, более простых, ни для современных, гораздо более сложных компиляторов до сих пор не предложено практических автоматизированных методов такой верификации, хотя поиск решений в этом направлении не прекращается [32].

Особенно остро проблема корректности встает при создании оптимизирующего двоичного транслятора. Для борьбы с ошибками в языковом компиляторе у разработчиков программ имеются средства, например, можно понизить уровень оптимизаций, поскольку менее оптимизирующие компиляторы, как правило, бывают существенно более надежными. Но двоичный транслятор полностью скрыт от пользователя, и его ошибки воспринимаются как ошибки в аппаратуре.

При верификации недостаточно доказать одну только семантическую корректность выполняемых оптимизирующих преобразований, хотя это является важнейшей частью верификации компилятора. Компиляторы используют в процессе оптимизаций сложнейшие структуры данных, обрабатываемые нетривиальными алгоритмами, от правильной работы которых существенно зависит корректность самих преобразований. Наконец, язык реализации также может вносить свои коррективы в надежность компилятора, поскольку для получения промышленных характеристик разработчики компиляторов используют языки типа Си или Си++, которые обладают рядом опасных свойств (зависшие ссылки, утечки памяти, неконтролируемое переполнение массивов и прочее). Еще одна опасность возникает, если при оптимизациях делается упор на декларированные свойства «правильных» языковых программ (например, несовпадение указателей на объекты разных типов в Си или несовпадение объектов, на которые смотрят разные параметры в Фортране), которые в действительности не соблюдаются программистами, и это отклонение от стандарта языка трудно или невозможно обнаружить. Все опасности подобного рода требуют специальных средств для их выявления.

При реализации ОКЭ и ОДТЭ используются более традиционные методы верификации [33]. Во-первых, в компиляторы встроены мощные средства внутреннего контроля, позволяющие выявлять ошибки при компиляции программ без запуска их на исполнение. Использование таких средств в сочетании со специально созданной системой тестов позволяет существенно повысить надежность оптимизирующих компиляторов. Достоинством данного метода является и то, что он с успехом может применяться при трансляции программ любого размера и сложности, повышая тем самым объем покрытия текста компилятора необходимыми проверками.

Во-вторых, специально созданные пакеты тестов позволяют при их исполнении сравнительно легко локализовать ошибки, не выявленные в процессе статической верификации. Эти пакеты постоянно пополняются двумя основными способами:

- созданием небольшого тестового примера из большой реальной задачи, на которой была обнаружена ошибка
- написанием специальных «направленных» тестов, проверяющих, корректно ли реализована функциональность компиляторов

В-третьих, созданы специальные средства для поиска и локализации ошибок исполнения, возникающих на больших задачах. Это наиболее сложная часть отладки компиляторов, поскольку зачастую проявлением ошибки являются неверные результаты работы программы. Во многих случаях процесс обнаружения таких ошибок трудно поддается автоматизации и требует больших интеллектуальных усилий. Его частичная автоматизация, используемая при отладке ОКЭ и ОДТЭ, позволяет существенно ускорить поиск наиболее сложных ошибок.

Наконец, для устранения ошибок, обусловленных языками реализации (в основном, это С и не-много С++) используются специальные верификаторы типа Purify [34] и собственный компилятор ОКЭ, работающий в так называемом защищенном режиме² [35, 36].

Заключение

В работе приводится анализ методов оптимизирующей компиляции, направленных на повышение скорости исполнения программ совместными аппаратно-программными средствами.

Оптимизирующие компиляторы для архитектурных платформ с явным параллелизмом, объединяющие в себе методы статической и динамической оптимизации, представляются наиболее перспективными. Они не только позволяют эффективно реализовать языковые программы, но обеспечивают также эффективность виртуальных платформ, таких как JVM или .NET CLI. Кроме того, они служат мощным средством обеспечения совместимости с наиболее распространенными аппаратными платформами, такими как IA-32, особенно при наличии аппаратной поддержки технологии двоичной трансляции. Оптимизирующие компиляторы с языков (ОКЭ) и оптимизирующая система двоичной трансляции (ОДТЭ), разработанные в рамках проекта по созданию микропроцессора Эльбрус-3М, относятся к этому новому типу. Благодаря мощным параллельным аппаратным ресурсам в статическом языковом компиляторе реализовано множество агрессивных оптимизаций, существенно повышающих производительность компилируемых программ. Благодаря фундаментальной аппаратной поддержке технологии двоичной трансляции оптимизирующий двоичный транслятор позволяет выполнять двоичные коды быстрее, чем на исходной аппаратной платформе.

Важными также следует считать многие новые алгоритмы и методы анализа и оптимизации программ, разработанные при реализации данных компиляторов.

Описанная в статье работа продолжается. Выполняется интенсивная «настройка» методов глобального планирования сложных регионов. Совершенствуются алгоритмы межпроцедурного анализа. Ведется поиск резервов в уже реализованных оптимизациях, как с точки зрения их упрочнения, так и с точки зрения более широкой применимости. Определяются новые методы оптимизации. Применяемые алгоритмы ускоряются или заменяются на новые, более быстрые. Совершенствуется технология двоичной трансляции, в частности, за счет улучшения профилирования, оптимизации накладных расходов на восстановление, введения фоновой (инкрементальной) оптимизации. Углубляются методы верификации и отладки компиляторов. В дальнейшем вся это и составит наиболее важные направления исследований технологии оптимизирующей компиляции.

Литература

1. K. Diefendorf "The Russians Are Coming: Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee" Microprocessor report, vol. 11, num. 2, Feb. 15, 1999.
2. Intel Corporation "Intel IA-32 Architecture Software Developer's Manual", Vol.1-3 2003.
3. P. N. Glaskowsky "IBM Raises Curtain on Power5". Microprocessor Report, vol.17, Archive 10, pp.13-14, October 2003.
4. L. Baraz, T. Devor, O. Etsion, S. Goldenberg, A. Skaletsky, Y. Wang and Y. Zemach "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based system." Proceeding of the 36th International Symposium on Microarchitecture (MICRO-36'03).
5. Intel Corporation. "Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization". Apr. 2003.
6. K. Krewell "Transmeta Gets More Efficient". Microprocessor Report, vol.17, Archive 10, pp.1,5-6, October 2003.
7. J. L. Hennessy, D. A. Patterson "Computer Architecture: A Quantitative Approach" Third Edition, Morgan Kaufmann publishers, 2003.
8. T. Lindholm, F. Yellin. "The Java Virtual Machine Specification". – Addison-Wesley, 1997

² Защищенная реализация языков С и С++ для базовой архитектуры – это еще одно уникальное свойство, которое реализовано в ОКЭ. Оно опирается на аппаратную поддержку и поддержку операционной системы и служит мощнейшим средством отладки программ. Но описание этой технологии выходит за рамки данной статьи.

9. Microsoft .NET Framework Developer Center “The Common Language Runtime (CLR)”. <http://msdn.microsoft.com/netframework/programming/clr/default.aspx>, Microsoft, 2004.
10. Muchnick S., “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers, 1997
11. Ахо А., Сети Р., Ульман Дж. “Компиляторы – принципы, технологии, инструменты”, Вильямс, 2001
12. E. R. Altman, D. Kaeli, Y. Sheffer “Welcome to the Opportunities of Binary Translation”. Computer, Vol.33, No. 3, pp. 40-45, March, 2000.
13. С.А.Рожков “Надежность оптимизирующих двоично-транслирующих систем”. Информационные Технологии и Вычислительные Системы, Москва 1,1999, с.14-22
14. Дроздов А.Ю., Корнев Р.М., Боханко А.С. “Индексный анализ зависимостей по данным”. Информационные технологии и вычислительные системы, этот номер
15. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, Roger A. Bringmann “Effective Compiler Support for Predicated Execution Using the Hyperblock” In Proceedings of the 25th International Symposium on Microarchitecture, pp.45-54, December 1992.
16. D. I. August, W. W. Hwu, and S. A. Mahlke “A Framework for Balancing Control Flow and Predication”. Proceedings of the 30th International Symposium on Microarchitecture, December, 1997.
17. Волконский В. Ю., Окунев С. К. “Предикатное представление как основа оптимизации программы для архитектур с явно выраженной параллельностью”. Информационные технологии, №4. Апрель, 2003.
18. Волконский В.Ю, Окунев С.К. "Оптимизация критического пути на предикатном представлении программы". Информационные технологии, № 9. Москва, сентябрь 2003.
19. Дроздов А.Ю., Степаненков А.М. “Технология оптимизации цикловых участков процедур в компиляторах для архитектур с аппаратной поддержкой конвейеризации циклов”. Информационные технологии и вычислительные системы, этот номер
20. S. Winkel "Optimal Global Scheduling for Itanium Processor Family". Second Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology, 2002.
21. Баскаков Ю.В., Волконский В.Ю., Грабежной А.В., Нейман-заде М.И., Тарасенко Л.Г., “Поддержка процесса повышения производительности компиляторов”. Информационные технологии и вычислительные системы, этот номер
22. H.-H. Lee, G. S. Tyson, M. Farrens “Eager Writeback – a Technique for Improving Bandwidth Utilization”. Proceedings of the 33th International Symposium on Microarchitecture, December, 2000.
23. T. M. Chilimbi, M. D. Hill, J. R. Larus “Making Pointer-Based Data Structures Cache Conscious” – IEEE Computer, Vol. 33, No. 12, Dec. 2000, pp. 67-74
24. Волконский В.Ю., Дроздов А.Ю., Ровинский Е.В. “Метод использования мелкоформатных векторных операций в оптимизирующем компиляторе”. Информационные технологии и вычислительные системы, этот номер
25. Останиевич А.Ю. “Экспериментальное исследование аппаратной поддержки предикатных вычислений в архитектуре с явно выраженным параллелизмом”. Ж-л Информационные технологии и вычислительные системы, 1, 1999.
26. S.A. Mahlke, R.E. Hank, J. McCormick, D.I. August and W.W. Hwu “A comparison of full and partial predicated execution support for ILP processors” In Proceedings of the 22th International Symposium on Computer Architecture, pp.138-150, June, 1995.
27. T. Ball, J.R. Larus “Branch Prediction For Free”. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.300-313, 1993.
28. Волконский В.Ю., Масленников Д.М., Ровинский Е.В. “Развитие метода статического предсказания профильной информации” – Высокопроизводительные вычислительные системы и микропроцессоры. Сборник научных трудов. Выпуск 2, стр. 3-20. Москва, 2001.
29. Василец П.С., Масленников Д.М., Драгошанский О.С. “Оптимизация обработки вещественного контекста в двоично-оптимизирующей системе”. Информационные технологии и вычислительные системы, этот номер
30. Дроздов А.Ю., Степаненков А.М. “Управляемые пакеты оптимизаций”. Информационные технологии и вычислительные системы, этот номер
31. Гимпельсон В.Д., Масленников Д.М., Волконский В.Ю. “Быстрый алгоритм минимизации высоты графа зависимостей”. Информационные технологии и вычислительные системы, этот номер
32. S. Lerner, T. Millstein, G. Chambers “Automatically Proving the Correctness of Compiler Optimizations”. PLDI’03, pp.220-231, June 9-11, 2003. San Diego, California, USA.
33. Лаврешников А.А., Рогов Р.Ю., Тарасенко Л.Г. “Система поддержки процесса разработки и выпуска версий программного комплекса”. Информационные технологии и вычислительные системы, этот номер
34. IBM Rational PurifyPlus
<http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/PurifyPlusPDF.pdf>
35. Волконский В.Ю., Тихонов В.Г., Эльцин Е.А. “Реализация языков программирования, гарантирующая межмодульную защиту” – Высокопроизводительные вычислительные системы и микропроцессоры. Сборник научных трудов. Выпуск 2, стр. 3-20. Москва, 2001.

36. Волконский В.Ю., Тихонов В.Г., Эльцин Е.А., Матвеев П.Г. “Реализация объектно-ориентированных языков программирования, гарантирующая межмодульную защиту” – Высокопроизводительные вычислительные системы и микропроцессоры. Сборник научных трудов. Выпуск 4, стр. 18-37. Москва, 2003.

Волконский Владимир Юрьевич. Родился в 1950 году. Окончил Московский государственный университет им. М.В. Ломоносова в 1972 году. Кандидат технических наук с 1980 года, автор 15 научных статей и 23 патентов. Область научных интересов – компиляторные технологии, микропроцессорные архитектуры, высокопроизводительные вычисления. Зав. лабораторией ИМВС РАН.