Чилингарова Софья Александровна

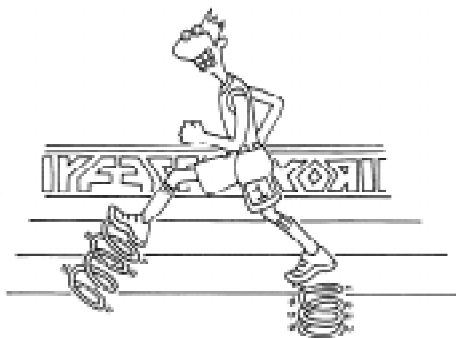
МЕТОДЫ ОПТИМИЗАЦИИ ДЛЯ ДИНАМИЧЕСКИХ (JUST-IN-TIME) КОМПИЛЯТОРОВ

Часть 1. Общие принципы и архитектура

Задача оптимизации для динамического (*just-in-time*) компилятора подобна задаче оптимизации для обычного статического компилятора с одним существенным исключением. Динамический компилятор работает во время выполнения программы, и время, затраченное на компиляцию, добавляется к общему времени выполнения, воспринимаемому пользователем. Применение сложных и дорогостоящих оптимизаций может существенно увеличить быстродействие и эффективность использования ресурсов для сгенерированного кода, но, вместе с тем, настолько оттягивать начало выполнения очередного компилируемого метода (или поглощать так много ресурсов), что общий воспринимаемый эффект будет нулевым или даже отрицательным. Таким образом, в оптимизационной задаче появляется дополнительное ограничение – время работы самого компилятора.

С другой стороны, генерация кода во время выполнения дает также и дополнительные преимущества. Во время выполнения доступна самая точная информация о целевой платформе (например, известен конкретный тип процессора, а не просто семейство процессоров), доступных ресурсах, среде выполнения, и, следовательно, возможна более точная настройка. Кроме того, мы можем непосредственно наблюдать реальное поведение программы, например, какие методы или блоки выполняются чаще, к каким именно объектам идет обращение при вызове виртуальных методов. Мы можем строить стратегию оптимизации с учетом сведений, собранных во время выполнения, вместо того чтобы применять сложные эвристики или пытаться смоделировать поведение программы во время статической компиляции.

Для сбора информации о поведении программы применяются различные техники профилирования. Например, в исполняемый код внедряется инструментарий – дополнительные инструкции, которые обновляют динамический профиль программы. Это достаточно дорогостоящий способ: дополнительные инструкции могут сильно замедлить выполнение кода. Существуют



Применение сложных и дорогостоящих оптимизаций может существенно увеличить быстродействие...

также и другие методы профилирования, которые мы подробно рассмотрим далее. Точная информация об архитектурной платформе известна во время запуска виртуальной машины, в этот момент она собирается и затем используется динамическим компилятором.

Можно ли (и нужно ли) применять в *just-in-time* компиляторах сложные и дорогостоящие методы оптимизации? Не лучше ли, из соображений скорости, ограничиться только простыми и быстрыми оптимизациями? Как лучше всего использовать преимущества компиляции во время выполнения? В чем и насколько может быть полезна собранная статистика поведения программы? Как не перегрузить код инструментарием и, вместе с тем, собрать достоверную информацию? В данной статье мы попробуем дать ответ на эти вопросы, проанализировав существующие научно-исследовательские проекты и коммерческие системы, в которых успешно применяются оптимизирующие динамические компиляторы.

После краткого обзора истории вопроса, мы рассмотрим общую архитектуру систем с оптимизирующими динамическими компиляторами, способы сбора статистики поведения программы во время выполнения, методы оптимизации, которые могут дать выигрыш в таких системах.

ИСТОРИЧЕСКАЯ СПРАВКА

Первыми широко известными виртуальными машинами были интерпретаторы языка LISP, разработанные в 50–60-х годах прошлого века. Эти машины включали в себя сборщик мусора и возможность динамической загрузки, а в 1962 году был создан динамический компилятор языка LISP.

Adaptive Fortran (1974) использовал большинство из тех приемов, которые на настоящий момент стали стандартом для динамических компиляторов и о которых пойдет речь ниже: выборочная оптимизация, профилирование (сбор статистики поведения программы), несколько уровней оптимизации, подсистема управления компиляцией и перекомпиляцией методов [9; 11].

В середине 80-х годов XX века были созданы виртуальные машины Self и Smalltalk, использовавшие динамические компиляторы с многоуровневой оптимизацией. Современные Java-машины (и их динамические компиляторы) вобрали в себя многие достижения этих виртуальных машин 80-х годов.

Широко распространенные сейчас виртуальные Java-машины появились в середине 90-х годов, а в 1995 году в научно-исследовательском центре IBM был создан первый динамический компилятор для языка Java в составе IBM DK for Java [1; 2]. Эта система активно развивается до настоящего времени и является одной из самых интересных разработок в области динамических компиляторов. Другой крупный проект, виртуальная машина Jalapeто (затем переименованная в Jikes RVM), написанная, в основном, на самом языке Java [3; 4] развивается с 1997 года. И Jikes, и IBM DK включают в себя подсистему динамической компиляции с многоуровневой оптимизаци-

CIL (Common Intermediate Language) – промежуточный язык виртуальных машин, определяемый спецификацией ECMA-335 (http://www.ecma-international.org/publications/standards/Ecma-335.htm). Этот язык используется, например, виртуальной машиной .NET, SSCLI (Rotor), Mono. См. *промежуточный код*.

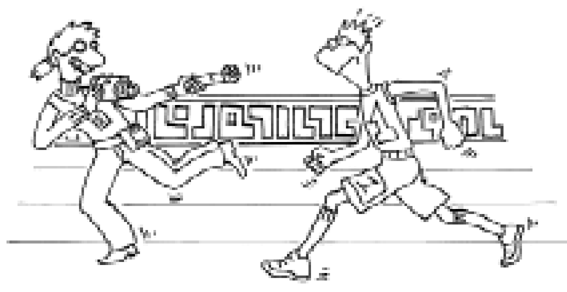
Inline-подстановка (inlining) – подстановка кода функции в место ее вызова. Эта операция может привести к увеличению быстродействия программы, как за счет избавления от расходов на вызов функции, так и за счет того, что inline-подстановка увеличивает поле для проведения других оптимизаций – код подставляемой функции может быть оптимизирован совместно с кодом вызывающей. Вместе с тем, неудачная inline-подстановка может привести к значительному росту объема кода, не увеличивая быстродействие.

Java-байткод – промежуточный язык виртуальной Java-машины. Определяется спецификацией от Sun (http://java.sun.com/docs/books/vmspec/index.html). См. *промежуточный код*.

ей и профилированием. Решения о компиляции и перекомпиляции методов принимаются на основе информации, собранной непосредственно во время выполнения программы. HotSpot JVM – современная виртуальная Java-машина от Sun [9] – использует два разных динамических компилятора: простой и быстрый вариант для клиентских приложений и сложный, применяющий множество разнообразных оптимизаций компилятор для серверных приложений [9]. Эти три проекта мы более подробно рассмотрим в последней части статьи. В настоящее время практически все коммерческие Java-машины используют оптимизирующие just-in-time компиляторы.

Платформа .NET, выпущенная в 2002 году Microsoft, использует динамический компилятор для трансляции в машинный код конструкций стандартного для этой платформы промежуточного языка CIL (Common Intermediate Language), на который, в свою очередь, может транслироваться код с самых разных языков программирования высокого уровня. Динамический компилятор платформы .NET производит над промежуточным кодом набор стандартных оптимизирующих преобразований [16; 17; 18].

Некоммерческий вариант .NET – платформа SSCLI (Rotor), созданная специально для академических исследований, включает только простой однопроходовый компилятор, без всяких оптимизаций [19]. В настоящее время уже есть успешные попытки внедрения оптимизирующих компиляторов в Rotor, также заслуживающие внимания [12; 14; 15]. В первой половине 2006 года ожидается выход в свет версии 2.0 SSCLI, которая будет содержать оптимизирующий компилятор.



Решения о компиляции и перекомпиляции методов принимаются на основе информации, собранной ...во время выполнения программы.

ОБЩИЕ ПРИНЦИПЫ ПОСТРОЕНИЯ СИСТЕМ С ОПТИМИЗИРУЮЩИМИ ДИНАМИЧЕСКИМИ КОМПИЛЯТОРАМИ

ВЫБОРОЧНАЯ КОМПИЛЯЦИЯ

Можно ли использовать в just-in-time компиляторах сложные оптимизации, отнимающие много времени? Практика дает следующий ответ: можно, если компилировать таким образом не все методы.

Общий принцип, которому следуют разработчики современных оптимизирующих динамических компиляторов, – выборочная компиляция наиболее часто исполняемого кода, которая производится, как правило, в несколько этапов. На каждом новом этапе применяются все более сложные и «дорогие» оптимизации. При первом вызове все методы либо интерпретируются, либо ком-

Агрессивные оптимизации – оптимизации, производимые на основе предположений, выведенных из данных профилирования, не подтвержденных точными результатами статического анализа. Иными словами, агрессивные оптимизации производятся на основе предположений, которые верны лишь с некоторой долей вероятности.

Алгоритм с линейным временем выполнения – алгоритм, время выполнения которого растет пропорционально числу входных параметров, то есть *время выполнения* = $O(n)$, где n – число входных параметров (см. http://en.wikipedia.org/wiki/Linear_time).

Базовый компилятор (в динамической компиляции) – компилятор, который компилирует все методы при их первом вызове. Как правило, не производит никаких или почти никаких оптимизаций. Если при первом вызове методы интерпретируются, место базового компилятора в системе занимает интерпретатор.

...во время выполнения выделяются так называемые «горячие» (hot) методы...

пилируются простым, очень быстрым компилятором, не применяющим вообще никаких оптимизаций или применяющим ограниченный набор самых «дешевых» оптимизаций. Затем с помощью профилирования непосредственно во время выполнения выделяются так называемые «горячие» (hot) методы – те, что вызываются наиболее часто или содержат циклы с большим числом итераций. Такие куски кода оптимизировать наиболее выгодно. Когда счетчик количества вызовов, или счетчик итераций цикла, или комбинация этих двух значений превышает установленное пороговое значение, метод компилируется (если при первых запусках для исполнения методов используется интерпретатор [1; 2]) или компилируется заново с более высоким уровнем оптимизации (ставится в очередь на повторную компиляцию), если используется простой (базовый) компилятор [3; 4].

Оптимизированный на первом этапе код также можно профилировать и, выделив новый набор еще более «горячих» методов, перекомпилировать их с более высоким уровнем оптимизации, то есть с использованием более сложных и, соответственно, требующих больше времени и ресурсов оптимизаций. Если предположение о том, что «горячие» методы будут в дальнейшем вызываться так же часто, как и во время профилирования, верно (а это, как правило, так, поскольку на момент получения статистики программа обычно работает уже в стабильном режиме), то даже относительно большие потери времени на компиляцию окупятся за счет суммарного выигрыша при многократном исполнении выбранных участков кода. Затем описанный процесс можно повторить еще раз и, выделив новый набор методов, перекомпилировать их с еще более высоким уровнем оптимизации. Таким образом, мы получаем высоко оптимизированный код для тех методов, выполнение которых занимает наибольшую часть времени, не перегружая при этом подсистему динамической компиляции (и следовательно, всю систему, работающую во время выполнения) задачами оптимизации кода, который выполняется редко. Время запуска также не увеличивается, так как при первом вызове все методы интерпретируются или компилируются очень быстро.

На практике обычно используются два или три уровня оптимизирующей компиляции (не считая базовой, неоптимизирующей компиляции или интерпретации). При большем количестве уровней, как показывает опыт, разница в эффективности с добавлением нового уровня становится намного менее существенной [1; 5]. Для хорошо

Девиртуализация – замена виртуальных вызовов прямыми. В случае, когда тип времени выполнения объекта, функции которого вызываются, можно вычислить, избавление от виртуального вызова помогает сократить расходы на поиск в таблице виртуальных методов и способствует дальнейшей оптимизации (например, может быть произведена inline-подстановка).

Динамическая компиляция – компиляция промежуточного кода виртуальной машины (Java-байткод, CIL) в машинный код во время выполнения.

Динамическое профилирование – производится в условиях эксплуатации системы, с целью выделения участков кода для последующие динамической компиляции.

Избыточные инструкции (избыточный код) – инструкции, которые можно удалить (с точностью до именования переменных) и от этого результат исполнения кода не изменится. Например, $a = b; c = a + d;$ можно заменить на $c = b + d$. Здесь $a = b$ – избыточный код.

сконфигурированной системы с двумя-тремя уровнями оптимизации, работающей в стабильном режиме, можно получить выигрыш в скорости в 2–5 раз по сравнению с тестовой ситуацией, когда все методы компилируются при первом вызове простым неоптимизирующим компилятором [1; 11]. Использование нескольких уровней оптимизации может дать выигрыш в производительности в 1,5–2 раза по сравнению с конфигурацией, включающей интерпретатор и один оптимизирующий компилятор для «горячих» методов, выполняющий только самые быстрые оптимизации.

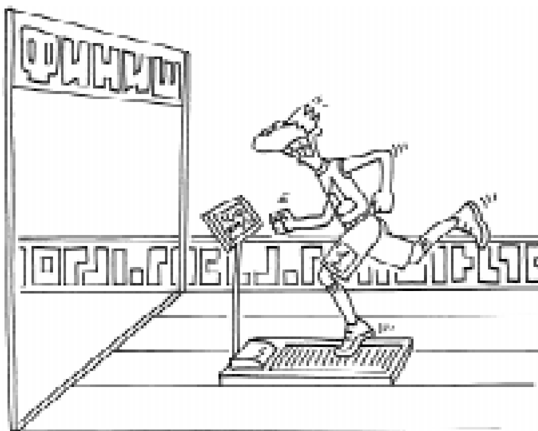
ПРОФИЛИРОВАНИЕ

Вопрос выбора техники профилирования также очень важен. Дополнительные инструкции, внедряемые в исполняемый код для сбора информации, увеличивают размер кода и могут сами по себе замедлить выполнение программы, если злоупотреблять ими на критических для скорости участках. Вместе с тем, собранная с помощью инструментария информация должна быть достаточно точной, чтобы на ее основе можно было принять правильное решение о компиляции. На практике используется несколько видов профилирования, и все эти техники, в целом, можно разбить на две группы: детерминированные и выборочные (sampling profiling).

В первом случае инструментарий внедряется непосредственно в код, генерируемый компилятором, или вызывается интерпретатором каждый раз, когда исполнение достигает определенных точек. Инструкции инструментария обычно вставляются на входе в метод и в местах, где выполняется переход назад, то есть там, где есть цикл. Они исполняются ровно столько раз, сколько исполняется инструментованный участок кода, и дают точную информацию. Эти инструкции производят очень простые действия – как правило, просто наращивают счетчики, но, так как частота их исполнения прямо пропорциональна частоте исполнения основного кода, наиболее чувствительным к перегрузке инструментарием оказываются наиболее часто исполняемые участки кода: часто вызываемые методы и короткие циклы с большим числом итераций. Это как раз те самые критичные участки, которые являются наиболее вероятными кандидатами на оптимизацию, информацию о которых важно собрать, и, вместе с тем, это те самые участки, где вставка инструментария сильнее всего сказывается на скорости работы программы.

Другой способ – выборочное профилирование. В этом случае с некоторой периодичностью (через некоторый фиксированный интервал времени или один раз за некоторое фиксированное число выполнений

Рисунок 1. Стандартная архитектура подсистемы динамической компиляции.



...делаются выборки информации о состоянии исполняемой программы...

данного участка кода) делаются выборки информации о состоянии исполняемой программы, и затем на основе этой выборочной информации принимаются решения. Информация в этом случае не является точной, но, если правильно подобрать частоту выборок, она достаточно хорошо (с достоверностью 95% и более) отражает реальную картину в ситуации стабильной работы [6; 8]. Обычная реализация этого метода такова: с некоторой периодичностью делается снимок состояния стека и регистров, по данным в регистре PC профайлер определяет, какому методу принадлежит инструкция, исполняемая в данный момент, а по снимку стека – кем был вызван этот метод. Здесь, как правило, не просто наращивается счетчик, но учитывается также динамика вызовов и составляется граф вызовов, который обновляется с каждой выборкой и используется при принятии решений о компиляции и *inline*-подстановках.

Существует вариация данного метода, когда не инструментованный код (исполняющийся большую часть времени) с некоторой периодичностью подменяется на небольшое время инструментованным кодом. Эта техника используется для сбора более подробной информации о поведении конкретных методов, которые предполагается оптимизировать [1], а также предлагается как альтернативный вариант общего профилирования с небольшой нагрузкой на систему [8].

В некоторых проектах для сбора статистики используются возможности процессора. Компилятор StarJIT, созданный в исследовательской лаборатории Intel, работающий с разными промежуточными языками (Java-байткод и CIL) и с несколькими семействами процессоров [12], в своей вариации для процессора Itanium полагается на данные о производительности, собранные аппаратно и сохраняемые в специальном разделе памяти процессора (PMU – Performance Monitoring Unit). Эту способность процессора Itanium использует и компилятор JRocket (BEA).

В современных виртуальных машинах детерминированное профилирование (внедрение постоянно действующего инструментария) обычно используется на первой стадии для неоптимизированного или интерпретируемого кода. Нагрузка, создаваемая инструментарием, не очень велика по сравнению со скоростью исполнения на этом этапе[1], реализация такого профилирования проста, что позволяет легко собрать нужную информацию и выбрать «кандида-

Инструментарий (в профилировании) – дополнительные инструкции, внедряемые в исполняемый код для измерения наблюдаемой величины (например, для регистрации времени выполнения или количества запусков). См. *Профилирование*.

Линейное время выполнения – см. алгоритм с линейным временем выполнения.

Пролог (процедуры или функции) – набор машинных инструкций, подготавливающих машину к выполнению процедуры или функции и сохраняющих состояние машины в момент вызова, необходимое для корректного возврата (например, сохранение значений регистров, указателя на вершину стека). Пролог вставляется компилятором в начало каждой процедуры или функции.

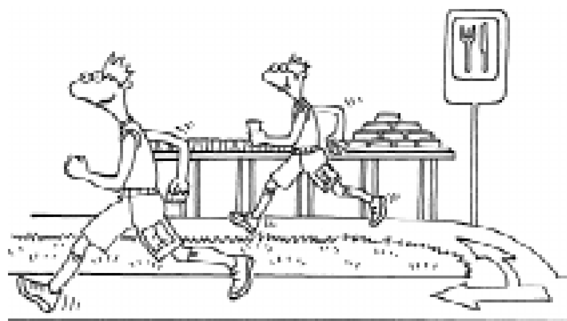
Промежуточный код (промежуточный язык виртуальной машины) – универсальный по отношению к операционной системе и аппаратной платформе язык, используемый для хранения кода, исполняемого виртуальной машиной. Промежуточный код (такой как *Java-байткод* или *CIL*) должен транслироваться в машинные коды (интерпретироваться или компилироваться) во время выполнения.

тов» на первую компиляцию. На более высоких ступенях оптимизации нагрузка, создаваемая постоянным инструментарием, оказывается неприемлемой, и используется выборочное профилирование.

АРХИТЕКТУРА ПОДСИСТЕМЫ КОМПИЛЯЦИИ

Общая схема архитектуры подсистемы динамической компиляции в современной виртуальной машине представлена на рисунке 1. Как правило, такая система состоит из контроллера, управляющего решениями о компиляции/перекомпиляции методов, одного базового неоптимизирующего компилятора или интерпретатора, 2-х или 3-х уровневого оптимизирующего компилятора, управляемого контроллером, профайлера, отвечающего за сбор данных о поведении программы, и хранилища данных профилирования, которые используются контроллером по мере необходимости.

Базовый компилятор/интерпретатор вызывается автоматически при обращении к не компилировавшемуся ранее методу. Если используется базовый компилятор, обычное решение таково – в точке входа в метод в структуре метаданных размещается адрес заглушки (stub), которая вызывает базовый компилятор и затем замещает свой адрес на адрес скомпилированного кода [7; 19]. Начало исполнения метода при первом вызове задерживается на время его компиляции, но, так как базовый компилятор работает быстро, эта задержка невелика и для пользователя не заметна.



Компиляция оптимизирующим компилятором производится в отдельном потоке...

Контроллер компиляции/перекомпиляции отслеживает изменения данных профилирования и принимает решения о компиляции определенных методов с определенным уровнем оптимизации. Компиляция оптимизирующим компилятором производится в отдельном потоке, не задерживая выполнение: до тех пор пока компилятор не закончит работу над новой оптимизированной версией, выполняется старая версия метода. Когда компиляция завершается, в точке входа в метод подставляется новый адрес, и при дальнейших вызовах используется уже новая, более высоко оптимизированная версия.

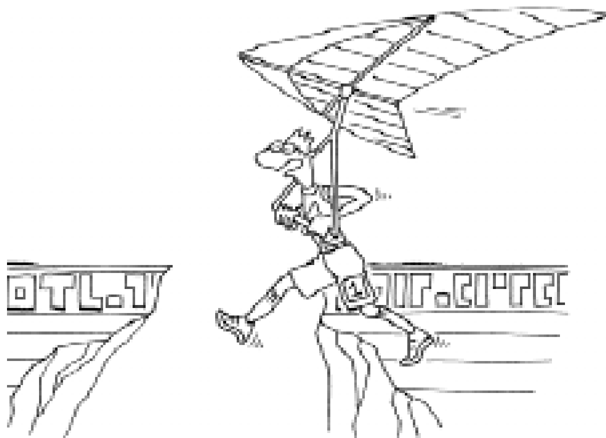
Если по какой-либо причине переход на новую версию кода желательно сделать раньше, чем завершится текущий вызов (например, метод содержит цикл с большим числом итераций), часто применяется механизм, называемый «замещением на стеке» (On Stack Replacement – OSR) [5]. Механизм этот работает так: выполнение метода при-

Профилирование – измерение различных параметров выполнения программы, например, скорости выполнения или количества запусков отдельных функций.

Распространение констант (constant propagation) – для переменных, значения которых – константы, подстановка этих константных значений вместо обращения к переменной в местах ее использования (например, *a = 4; b = b + a;* заменяется на *b = b + 4;*). Этот вид оптимизации уменьшает число машинных инструкций в исполняемом коде на выходе компилятора (у многих процессоров есть специальные команды для операций с константами) и особенно полезен в сочетании со *сверткой констант*.

Регистр PC (Program Counter) или IP (Instruction Pointer) – обобщенное название для регистра, в котором хранится указатель на выполняемую в данный момент инструкцию.

Регистровая архитектура – архитектура, в которой операции выполняются в регистрах. Такова архитектура многих современных физических машин.



На самом высоком уровне применяются наиболее сложные оптимизации, которые могут дать выигрыш только на особенно критичных, часто исполняемых участках...

останавливается в одной из определенных заранее «безопасных» точек, затем запускается специально для данного метода сгенерированный переходный код, который сохраняет значения локальных переменных, загружает эти значения туда, где их будет ожидать новая версия метода, и совершает безусловный переход на соответствующую инструкцию оптимизированного кода.

Методы, выбранные для компиляции с более высоким уровнем оптимизации, обычно помещаются в очередь, откуда их затем извлекает компилятор, работающий в отдельном потоке. Часто компиляция производится в нескольких параллельных потоках, в каждом из которых работает свой экземпляр компилятора.

Профайлер управляет сбором информации о поведении программы. Если неоптимизированный или первоначально интерпретируемый код уже содержит инструкции, регистрирующие нужные данные, то для сбора информации об оптимизированных методах (либо если от жесткого внедрения инструментария отказываются вообще) приходится предпринимать специальные действия: периодически подставлять инструментованный код, делать снимки состояния стека, регулировать частоту обращения для получения выборочных данных, инструментировать на короткие промежутки времени методы, выбранные для дальнейшей оптимизации. За все эти действия отвечает профайлер. Профайлер внедряет инструментарий в код и размещает в памяти объекты, делающие снимки стека и регистров. Собранные данные сохраняются в хранилище данных профилирования, откуда они затем извлекаются контроллером и используются при составлении плана компиляции. Работа профайлера также обычно управляется контроллером, который составляет план профилирования, в зависимости от конкретной ситуации и параметров, заданных пользователем.

ОПТИМИЗАЦИИ, ПРИМЕНЯЕМЫЕ НА РАЗНЫХ УРОВНЯХ

Решение о компиляции метода с определенным уровнем оптимизации принимается на основе следующего соотношения: сумма времени компиляции и ожидаемого времени выполнения оптимизированного метода должна быть больше ожидаемого времени выполнения не оптимизированного кода. Чем выше уровень оптимизации, тем для меньшего числа методов выполня-

Сборщик мусора (Garbage Collector) – механизм, предусмотренный в динамической среде выполнения (виртуальной машине) для предотвращения утечек памяти. В среде, где есть сборщик мусора, динамически выделяемая память не должна освобождаться «вручную». Сборщик мусора периодически проверяет, какие объекты в динамической памяти на данный момент «живые» (то есть, существуют ссылки на эти объекты либо из исполняемого в данный момент кода, либо из других «живых» объектов), и удаляет все остальные.

Свертка констант (constant folding) – вычисление выражений, в которых все операнды – константы, во время компиляции, а не во время выполнения.

Снимок состояния стека (stack sample) – выборка данных, содержащая информацию о том, активационные записи каких методов находятся в данный момент на стеке (то есть какие методы на данный момент были вызваны в данном потоке), в порядке вложенности вызовов.

ется это соотношение. На первом, самом низком уровне, применяются наиболее простые, быстрые и гарантированно полезные в большинстве случаев оптимизации. На самом высоком уровне применяются наиболее сложные оптимизации, которые могут дать выигрыш только на особенно критичных, часто исполняемых участках, и только при специфических условиях. Если на первом уровне собственно при компиляции данные профилирования, как правило, не используются (превышение порога счетчиком инициирует компиляцию, но сама компиляция производится так же, как и статическая, только с ограничением на сложность оптимизаций), то на более высоких уровнях данные профилирования используются очень активно, и некоторые распространенные оптимизации (такие, как inline-подстановка) просто невозможны без данных профилирования.

В первых работах, посвященных многоуровневым динамическим компиляторам, наборы оптимизаций, проводимых на разных уровнях, различались существенно [1; [3; 15]. Однако в настоящий момент практически все успешные проекты демонстрируют примерно одинаковую схему «расстановки» оптимизаций по уровням. Таким образом, можно говорить о том, что в настоящее время опытным путем сформировалось общее представление о том, какие виды оптимизаций следует применять на разных уровнях.

Наиболее популярные в случае динамических компиляторов оптимизации – это удаление избыточных присваиваний и inline-подстановки. Стековая архитектура современных виртуальных машин предполагает, что все данные, для того чтобы их можно было использовать, должны быть загружены в стек. Локальные переменные, переменные-члены классов, ссылки на объекты и методы хранятся в специально отведенных для них областях памяти и загружаются на стек, когда нужно произвести какие-либо операции над ними, затем сразу выгружаются обратно. При отображении на архитектуру реальных компьютеров эта схема приводит к большому числу избыточных копирований, загрузок и сохранений в память. Рассмотрим пример. Возьмем следующий простой метод на языке C#:

```
class Count
{
 ....
 public int div10000(int a)
 {
 int divider = 3;
 int b = a;
 for (int j = 0; j < 10000; j++)
 {
 b = b/divider;
 }
 return b;
 }
}
```

В таблице 1 приведен код на промежуточном языке CIL, сгенерированный для этого метода компилятором C#, и машинный код для архитектуры x86, сгенерированный JIT-компилятором SSCLI, не применяющим почти никаких оптимизаций. JIT-компилятор SSCLI отображает абстрактный стек виртуальной машины на физи-

Статическая компиляция – традиционный способ компиляции и сборки модулей программы, когда компиляция и выполнение программы разделены во времени. Конструкции исходного языка компилируются в машинный код во время разработки, и затем во время выполнения в память машины загружаются уже готовые последовательности машинных инструкций.

Статическое профилирование – производится во время разработки, для улучшения параметров системы. Для профилирования создается специальная инструментированная версия кода. См. *Профилирование*.

Стековая архитектура – архитектура, в которой все операции выполняются на стеке: операнды берутся с вершины стека (куда они предварительно загружаются из памяти, либо помещаются в результате предыдущих операций), результат операции кладется на стек. Стековую архитектуру в настоящее время используют, в основном, виртуальные машины, такие как Java-машина или .NET.

ческий стек, вершину стека держит в регистре eax. По стандартному соглашению CIL (CIL Calling Convention) аргументы передаются через стек, возвращаемое значение также сохраняется на стеке. JIT-компилятор SSCLI реализует это соглашение для аргументов буквально, а возвращаемое значение, если оно помещается в регистр, передает через регистр **eax** – таким образом оно оказывается на вершине абстрактного стека, после того как вызывающий метод сбрасывает стек.

Как мы видим, в машинном коде очень много избыточных операций. Рассмотрим, как можно оптимизировать его путем удаления избыточных копирований и распространения констант. Введем промежуточное представление, отражающее все операции, генерируемые однопроходовым JIT-компилятором. Обозначим регистры буквами **r** с номерами, аргументы – **a**, локальные переменные – **l**. Обозначим через **r1** регистр, в который помещается вершина стека (**eax**).

В таблице 2 показана последовательность преобразований, которые можно произвести над промежуточным кодом. В первой колонке – не оптимизированное промежуточное представление, построенное в соответствии с правилами, применяемыми JIT-компилятором SSCLI. Базовые блоки, разделены пустыми строками. На самом деле уже во время генерации промежуточного представления можно произвести удаление избыточных копий на уровне базовых блоков – результат представлен во второй колонке. В третьей колонке – результат рас-

Таблица 1. **Код, сгенерированный неоптимизирующим JIT-компилятором**

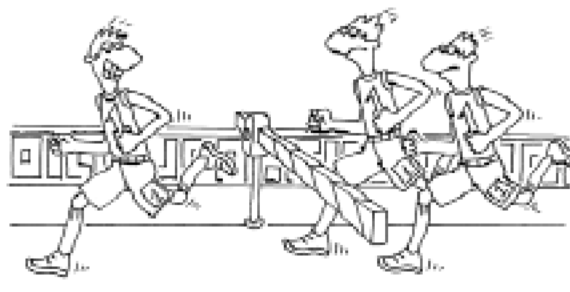
<table>
<tr><th>Код метода на языке CIL</th><th>Код, сгенерированный JIT-компилятором SSCLI</th></tr>
<tr><td><pre>
.maxstack 2
.locals init (int32 V_0,
        int32 V_1,
        int32 V_2)
IL_0000: ldc.i4.3
IL_0001: stloc.0
IL_0002: ldarg.1
IL_0003: stloc.1
IL_0004: ldc.i4.0
IL_0005: stloc.2
IL_0006: br.s   IL_0010
IL_0008: ldloc.1
IL_0009: ldloc.0
IL_000a: div
IL_000b: stloc.1
IL_000c: ldloc.2
IL_000d: ldc.i4.1
IL_000e: add
IL_000f: stloc.2
IL_0010: ldloc.2
IL_0011: ldc.i4   0x2710
IL_0016: blt.s   IL_0008
IL_0018: ldloc.1
IL_0019: ret
</pre></td><td><pre>
02D82EC4 mov   ecx,4
02D82EC9 push 0
02D82ECB loop 02D82EC9
02D82ECD mov   eax,3
02D82ED2 mov   dword ptr[ebp-14h],eax
02D82ED5 mov   eax,dword ptr [ebp+0Ch]
02D82ED8 mov   dword ptr [ebp-18h],eax
02D82EDB xor   eax,eax
02D82EDD mov   dword ptr [ebp-1Ch],eax
02D82EE0 mov   eax,dword ptr [ebp-1Ch]
02D82EE3 push eax
02D82EE4 mov   eax,2710h
02D82EF2 pop   ecx
02D82EF3 cmp   ecx,eax
02D82EF5 jge   02D82F23
02D82EFB mov   eax,dword ptr [ebp-18h]
02D82EFE push eax
02D82EFF mov   eax,dword ptr [ebp-14h]
02D82F02 mov   ecx,eax
02D82F04 pop   eax
02D82F05 mov   edx,eax
02D82F07 sar   edx,1Fh
02D82F0A idiv eax,ecx
02D82F0C mov   dword ptr [ebp-18h],eax
02D82F0F mov   eax,dword ptr [ebp-1Ch]
02D82F12 push eax
02D82F13 mov   eax,1
02D82F18 pop   ecx
02D82F19 add   eax,ecx
02D82F1B mov   dword ptr [ebp-1Ch],eax
02D82F1E jmp   02D82EE0
02D82F23 mov   eax,dword ptr [ebp-18h]
02D82F3A mov   esi,dword ptr [ebp-4]
02D82F3D mov   esp,ebp
02D82F3F pop   ebp
02D82F40 ret
</pre></td></tr>
</table>

пространения констант. Далее – поместим локальные переменные **l2** и **l3** в регистры. Переименуем **r3** в **r1**, а **r4** – в **r3**. В последней строчке мы видим 9 операций вместо 28-ми, с использованием 3-х регистров. Еще меньше их получится, если передавать часть аргументов через регистры (как обычно и делается), в частности, первый – через регистр **r1**.

И действительно, компилятор .NET 1.1 генерирует для приведенного метода следующий код:

```
06CC00C5 mov ecx,3
06CC00CA cdq
06CC00CB idiv eax,ecx
06CC00CD inc esi
06CC00CE cmp esi,2710h
06CC00D4 jl 06CC00C5
06CC00D6 pop esi
06CC00D7 ret
```

Удаление избыточных копий, распространение и свертка констант на уровне базовых блоков и расширенных базовых бло-



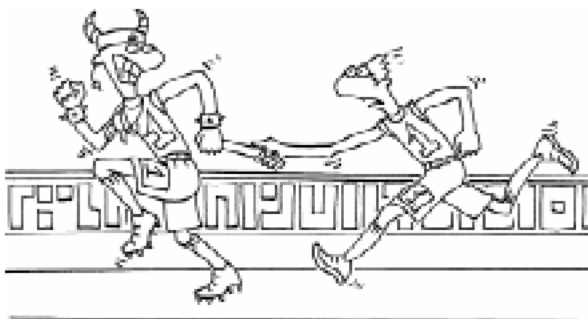
Удаление избыточных копий...

ков[1] не требует много времени и поэтому всегда производится уже на первом уровне оптимизации. Для распределения регистров на первом уровне обычно используются простые алгоритмы с линейным временем выполнения. Обязательно производится inline-подстановка «очень маленьких» методов – тех методов, размер которых (без пролога и эпилога) меньше размера кода, необходимого для вызова метода. Если при вызове виртуального метода объект, метод которого вызывается, точно известен, вы-

Таблица 2. **Последовательность оптимизирующих преобразований**

Промежуточное представление	Удаление копий	Распространение констант	r3 <= l2 r4 <= l3	r1 <= r3 r3 <= r4
00 r1 <- 3	00 l1 <- 3	00 l2 <- a1	00 r3 <- a1	00 r1 <- a1
01 l1 <- r1	01 l2 <- a1	01 l3 <- 0	01 r4 <- 0	01 r3 <- 0
02 r1 <- a1	02 l3 <- 0			
03 l2 <- r1		02 r2 <- l3	02 cmp r4, 0x2710	02 cmp r02,0x2710
04 r1 <- 0	03 r1 <- 0x2710	03 cmp r2, 0x2710	03 jge 08	03 jge 08
05 l3 <- r1	04 r2 <- l3	04 jge 13		
	05 cmp r2, r1		04 r2 <- 3	04 r2 <- 3
06 r1 <- l3	06 jge 16	05 r2 <- 3	05 idiv r3, r2	05 idiv r1, r2
07 s1 <- r1		06 r1 <- l2	06 add r4, 1	06 add r3, 1
08 r1 <- 0x2710	07 r2 <- l1	07 idiv r1, r2	07 jmp 02	07 jmp 02
09 r2 <- s1	08 r1 <- l2	08 l2 <- r1		
10 cmp r2,r1	09 idiv r1, r2	09 r2 <- l3	08 r1 <- r3	08 ret
11 jge 26	10 l2 <- r1	10 add r2, 1	09 ret	
	11 r1 <- 1	11 l3 <- r2		
12 r1 <- l2	12 r2 <- l3	12 jmp 02		
13 s1 <- r1	13 add r1, r2			
14 r1 <- l1	14 l3 <- r1	13 r1 <- l2		
15 r2 <- r1	15 jmp 03	14 ret		
16 r1 <- s1				
17 idiv r1, r2	16 r1 <- l2			
18 l2 <- r1	17 ret			
19 r1 <- l3				
20 s1 <- r1				
21 r1 <- 1				
22 r2 <- s1				
23 add r1, r2				
24 l3 <- r1				
25 jmp 06				
26 r1 <- l2				
27 ret				

[1] Расширенный базовый блок – последовательность базовых блоков, ни у одного из которых, за исключением первого, не может быть нескольких предшественников.



На втором уровне применяются более агрессивные inline-подстановки...

полняется девиртуализация – замена динамических вызовов для виртуальных методов статическими. Статические вызовы намного проще динамических (не нужно производить поиск в таблице виртуальных методов), и, когда информация о типе объекта легко доступна, этот вид оптимизации является одновременно простым и эффективным.

Также на первом уровне производятся некоторые специфичные для объектно-ориентированных языков оптимизации, такие как удаление избыточных проверок на NULL, проверок на выход за границы массивов, на исключения, избыточных приведений типов. На первом уровне такие преобразования, как правило, производятся «на лету», при переводе кода промежуточного языка (CIL или байткод) в простое внутреннее промежуточное представление, и ограничиваются диапазоном базового блока или расширенного базового блока. Иногда производятся также и некоторые глобальные преобразования: распространение и свертка констант, удаление избыточных проверок, но с очень строгими ограничениями на время работы и количество итераций.

Внутреннее промежуточное представление, используемое для оптимизации на первом этапе, – это, как правило, тот же промежуточный язык, дополненный операциями явных проверок на NULL, границы массива и исключения, а также операциями явного преобразования типов везде, где такие операции должны производиться виртуальной машиной во время выполнения.

На втором уровне применяются более агрессивные inline-подстановки не только для «очень маленьких», но и для более крупных методов. При определении стратегии таких подстановок активно используются данные, собранные с помощью профилирования. Если какой-либо метод чаще всего вызывается одним определенным методом, то, в случае превышения порогового значения счетчиком, для вызываемого метода может быть принято решение о компиляции вызывающего метода с inline-подстановкой вызываемого. Девиртуализация производится не только в тех случаях, когда тип объекта можно определить точно, но также и в тех, когда тип можно определить с большой долей вероятности на основе информации, собранной во время профилирования. Если затем в некоторых случаях принятое решение оказывается неверным, реализуются различные сценарии деоптимизации и отката, речь о которых пойдет ниже.

Распространение и свертка констант, распространение данных о типах объектов и проверок на NULL, границы массивов и исключения, производятся глобально, на уровне всего метода. Для выявления констант и определения времени жизни значений переменных используется глобальное SSA (Static Single Assignment)-представле-

Уровни оптимизации (в динамической компиляции) – в динамической среде часто используется сразу несколько вариантов компиляции, которые различаются по количеству применяемых оптимизаций (и соответственно, по времени компиляции – чем больше оптимизаций, и чем они сложнее, тем больше времени затрачивается на компиляцию). Чем больше ресурсов тратится на оптимизацию, тем выше *уровень оптимизации.*

Эпилог (процедуры или функции) – набор машинных инструкций, возвращающих машину в состояние, в котором она может продолжать выполнение вызывающей процедуры или функции после завершения вызываемой (например, восстановление значений регистров, сброс стека). Эпилог вставляется компилятором в конец каждой процедуры или функции.

ние[2]. На основании анализа потока данных, сделанного с помощью этого представления, применяются такие методы глобальной (уровня всего метода) оптимизации, как удаление неиспользуемого кода и перемещение кода (например, вынесение инициализации переменной за границы цикла). Распределение регистров также производится глобально, и здесь могут применяться более сложные алгоритмы, такие как раскраска графа.

SSA-представление может использоваться для анализа потока данных также и на первом уровне, если разработчик предпочитает придерживаться единого внутреннего представления для всех уровней оптимизации. При этом на первом уровне производятся только самые простые оптимизации, такие как распространение констант, информации о типе, и результатах проверки на NULL, и устанавливаются очень жесткие ограничения на количество итераций. На втором уровне применяются более сложные оптимизации и допускается больше итераций. Ограничения тем не менее остаются, так как компилятор должен уложиться в бюджет времени и ресурсов, отведенный ему во время выполнения.

На третьем уровне (если он есть) включаются наиболее дорогостоящие и специализированные методы оптимизации. Сюда входят замена полей объектов и элементов массива скалярными переменными, размещение на стеке (а не в «куче») объектов, которые используются только локально внутри метода, различные специальные оптимизации циклов. Последние могут включать введение двух версий цикла: оптимизированного для тех случаев, когда известно, что используемые данные корректны, и общей – для тех случаев, когда возможны исключения, раскрутку цикла (unrolling) – объединение нескольких итераций в одну, вынесение первой или последней итерации за границы цикла (peeling). Анализ потока данных производится с помощью SSA-представления, обычно так же, как и на втором уровне, но допускается большее число итераций. Также могут создаваться специализированные версии методов для часто повторяющихся ситуаций – например, для определенного сочетания входных параметров или значений переменных.

Если используется только два уровня оптимизирующей компиляции, то второй считается уровнем полной оптимизации, и некоторые из стандартных оптимизаций третьего уровня выполняются на втором. В частности, выполняется замена объектов скалярами и оптимизация циклов.

ДЕОПТИМИЗАЦИЯ

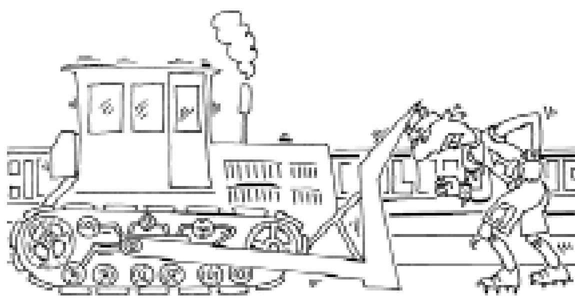
В динамических компиляторах, начиная со второго уровня оптимизации, широко применяются агрессивные (или оптимистичные) оптимизации. Метод оптимизируется под наиболее часто реализуемый сценарий (или с большой вероятностью единственный), выделенный с помощью профилирования или статических эвристик так, будто этот сценарий будет выполняться всегда, без учета других вариантов. В частности, сюда относятся агрессивные inline-подстановки, девиртуализация (когда тип объекта, метод которого вызывается, известен с большой долей вероятности), оптимистичная оптимизация обработки исключений. Если во время выполнения реализуется нетипичный сценарий, такой оптимизированный код ста-

[2] SSA (Static Single Assignment)-представление – это представление, в котором каждому значению переменной или элементарного выражения ставится в соответствие уникальное имя. Присваивание значения каждому имени делается только один раз.

Например, для выражения $a = a + b + c$ SSA-представление будет таким: $b_2 = b_1 + c_1$
$a_2 = a_1 + b_2$.

Если значение может быть инициализировано в нескольких местах, в зависимости хода выполнения программы, например: *if(...) { a=x; } else { a=y; } b=a;* то в месте соединения ветвей вводится новое имя, которое инициализируется φ-функцией: $a_3 = \varphi(a_1, a_2)$

SSA-представление полезно, когда нужно отследить движение конкретных значений, чтобы оптимизировать процесс их вычисления и хранения. Например, это представление часто используется для определения времени жизни значений и распределения регистров, удаления неиспользуемого кода (присваивания, которые нигде не используются), удаления избыточного копирования и избыточных вычислений.



Необходимо предусмотреть способы оперативного отката...

новится невалидным. Необходимо предусмотреть способы оперативного отката агрессивных оптимизаций в нетипичных ситуациях. Такие действия называются деоптимизацией. Существует несколько вариантов решения этой проблемы.

Самый простой – введение защищенных (guarded) участков кода. Перед оптимизированным участком вставляется проверка, и, если реальные условия соответствуют предполагаемым, работает оптимизированная ветвь, если нет, – неоптимизированная. Недостаток этого способа в том, что слияние двух ветвей после защищенного участка инвалидирует всю информацию, верную только для часто исполняемой оптимизированной ветви (такую как данные о типах объектов или о том, что некое значение не null), и мы не можем пользоваться этой информацией для оптимизации оставшейся части метода.

Один из популярных методов – замещение на стеке (On Stack Replacement – OSR). Этот способ широко используется в Jikes [5] и в HotSpot JVM [9]. Реализуется он следующим образом. В тот момент, когда выявляется невалидность предположения (например, тип объекта не соответствует ожидаемому или бросается исключение), выполнение приостанавливается, затем генерируется и запускается специальный адаптационный код (специальный пролог – в Jikes, адаптер – в HotSpot), который сохраняет значения локальных данных и загружает их туда, где их будет ожидать неоптимизированная версия метода. После того как отрабатывает адаптер, неоптимизированная версия запускается с нужного места. Тот же механизм используется, если динамическая загрузка классов (возможное, но относительно редкое событие) делает невалидными предположения о времени компиляции. В этом случае все потоки, выполняющие метод, для которого нужно произвести деоптимизацию, останавливаются в одной из заранее определенных безопасных точек, после чего запускается механизм замещения на стеке. Этот же механизм часто используется для замены неоптимизированного кода оптимизированным прямо во время работы метода, что может быть полезно, если метод содержит цикл с большим числом итераций.

Таблица 3. **Пример работы механизма замещения на стеке**

Источник: Stephen J.Fink and Feng Qjan. Design, Implementation, and Evaluation of Adaptive Recompilation with On-Stack Replacement, IBM T.J. Watson Research Center, March 2003[5], Figure 1.

Код метода на Java	Байткод	Дескриптор локальной области	Специальный пролог

class C
{
static int sum(int c)
{
int y = 0;
for (int i=0; i<c; i++)
{
y += i;
}
return y;
}
}
	0 iconst_0
	1 istore_1
	2 iconst_0
	3 istore_2
	4 goto 14
	7 iload_1
	8 iload_2
	9 iadd
	10 istore_1
	11 iinc 2 1
	14 iload_2
	15 iload_0
	16 if_icmplt 7
	19 iload_1
	20 ireturn
		running thread : MainThread
		frame pointer : 0xSomeAddress
		program counter : 16
		local variables : L0(c) = 100;
		L1(y) = 1225; L2(i) = 50;
		stack expressions : S0 = 50;
		S1 = 100;
			ldc 100
			istore_0
			ldc 1225
			istore_1
			ldc 50
			istore_2
			ldc 50
			ldc 100
			goto 16
			0 iconst_0
			...
			16 if_icmplt 7
			...
			20 ireturn

Рассмотрим, как работает замещение на стеке на примере Jikes[5]. В таблице 3 приведен код простого метода на языке Java и байткод для этого метода. Предположим, во время работы метода возникла ситуация, когда нужно произвести замещение на стеке – например, некое предположение стало невалидным (что, конечно, сложно представить в данном конкретном случае, однако если, например, окружить оператор сложения внутри цикла проверкой какого-нибудь условия, ситуация станет вполне реальной) или, наоборот, сгенерирована новая оптимизированная версия. Тогда выполнение метода приостанавливается, виртуальная машина Jikes сохраняет *дескриптор локальной области* (scope descriptor), и по данным этого дескриптора генерируется специальный пролог, с которым и запускается новая версия метода.

Другой вариант – динамическое исправление кода (code patching) используется в компиляторе IBM DK [7]. Данная техника использует то свойство, что невалидность предположения часто можно определить заранее – например, когда при вызове метода передается аргумент типа, отличного от предполагаемого. Тогда соответствующий участок кода вызываемого метода замещается в памяти неоптимизированной последовательностью инструкций, подходящей для общего случая. Достоинство этого способа в том, что он просто реализуется и не требует производить так много действий во время выполнения, как замещение на стеке, недостаток – в том, что он применим только в ограниченном числе случаев [7].

ЗАКЛЮЧЕНИЕ

Мы рассмотрели стандартную архитектуру подсистемы динамической компиляции, которая включает в себя:

– контроллер, управляющий решениями о компиляции/перекомпиляции,

– профайлер, собирающий динамические данные о поведении программы и поставляющий их контроллеру,

– оптимизирующий компилятор, компилирующий методы, выбранные контроллером на указанном уровне сложности оптимизаций,

– интерпретатор или очень простой базовый компилятор, не производящий оптимизаций, который используется при первом вызове методов.

Профайлер может использовать стандартные техники инструментирующего профилирования для неоптимизированного или интерпретируемого кода или для сбора более точной информации о поведении методов, уже выбранных для повторной компиляции, но для относительно высоко оптимизированного кода такие техники слишком тяжеловесны, поэтому используется выборочное профилирование.

На сегодняшний день уже можно выделить эмпирически сформировавшийся набор оптимизаций, применяемых на разных уровнях сложности оптимизирующей компиляции, но о каких-то устоявшихся канонах в этой области говорить еще рано. В части 2 настоящей статьи мы подробно рассмотрим примеры реализации подсистемы оптимизирующей динамической компиляции в современных коммерческих и научно-исследовательских виртуальных машинах.

Литература

[1] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2001.

[2] T. Suganuma , T. Ogasawara , K. Kawachiya , M. Takeuchi , K. Ishizaki , A. Koseki , T. Inagaki, T. Yasue , M. Kawahito , T. Onodera , H. Komatsu , T. Nakatani. Evolution of a java just-in-time compiler for IA-32 platforms, IBM Journal of Research and Development, v.48 n.5/6, p.767-795, September/November 2004

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, J. Whaley. The Jalapeno Virtual Machine. IBM Systems Journal, vol. 39, No 1, 2000.

[4] David Grove and Michael Hind. The Design and Implementation of the Jikes RVM Optimizing Compiler. OOPSLA ’02 Tutorial, Nov 5, 2002

[5] Stephen J.Fink and Feng Qjan. Design, Implementation, and Evaluation of Adaptive Recompilation with On-Stack Replacement, IBM T.J. Watson Research Center, March 2003.

[6] J. Whaley. A portable sampling-based profiler for Java virtual machines. In ACM 2000 Java Grande Conference, June 2000.

[7] K. Ishizaki, M. Kawahito, T. Yasue, and H. K. and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct. 2000.

[8] M. Arnold. Online Profiling and Feedback-Directed Optimization of Java. PhD thesis, Rutgers University, October 2002.

[9] The Java HotSpot▪ Virtual Machine, v1.4.1, d2, A Technical White Paper. Sun Microsystems, September 2002.

[10] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In USENIX Java Virtual Machine Research and Technology Symposium, pages 1–12, 2001.

[11] Stephen Fink, David Grove, and Michael Fink. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM T.J. Watson Research Center, 2004.

[12] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, Tatiana Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. Intel Technology Journal, vol. 07, Issue 01, February, 2003.

[13] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Vijay Menon, Brian R. Murphy, Tatiana Shpeisman. The StarJIT Dynamic Compiler – A Performance Study on the Itanium Architecture. 2nd Workshop on Managed Runtime Environments, 2004 (MRE’04).

[14] Todd Anderson, Marsha Eng, Neal Glew, Brian Lewis, Vijay Menon, and James Stichnoth. Experience Integrating a New Compiler and a New Garbage Collector into Rotor. Journal of Object Technology, Vol. 3, No. 9, 2004.

[15] Kapil Vaswani, Y.N. Srikant. Dynamic Recompilation and Profile-Guided Optimizations for a .NET JIT Compiler. IEE Software 2004.

[16] Kang Su Galtin. Power Your App with the Programming Model and Compiler Optimizations of Visual C++. MSDN Magazine, January 2005.

[17] Emmanuel Schanzer. Performance Considerations for Run-Time Technologies in the .NET Framework. MSDN Library, August 2001.

[18] Gregor Noriskin. Writing High-Performance Managed Applications: A Primer. MSDN Magazine, June 2003

[19] David Stutz, Ted Neward, Geoff Shilling. Shared Source CLI Essentials. O’Reilly, 2003.

Наши авторы, 2006.
Our authors, 2006.

Чилингарова Софья Александровна,
аспирант кафедры «Информатика»
математико-механического
факультета СПбГУ.