Обзор методов отладки оптимизированного кода

И. С. Кузнецов
KuznIvan@newmail.ru

В статье выполнен обзор работ, посвященных решению проблемы отладки оптимизированного кода. Представлены два основных направления исследований — обеспечение ожидаемого поведения и обеспечение истинного поведения. Изложены базовые идеи каждого направления, дано краткое описание решений, использованных в конкретных реализациях.

Введение

Практически все современные отладчики не поддерживают отладку оптимизированного кода. Однако иметь такую возможность во многих случаях желательно, а иногда и просто необходимо. Так, например, зачастую программа может требовать оптимизации уже лишь для того, чтобы быть хотя бы просто запускаемой на целевой машине, так как неоптимизированная версия может не помещаться в памяти или не успевать обрабатывать поток входных данных. Не менее важно уметь отлаживать оптимизированный код и в случае нахождения пользователями ошибок в уже выпущенном программном продукте. Так как финальная версия программы обычно компилируется с применением всех возможных оптимизаций, то и ошибки в данном случае следует искать, отлаживая именно оптимизированный код.

Оптимизированная и неоптимизированная программы не всегда ведут себя одинаково. Причин такого несоответствия может быть несколько. Традиционно программисты в подобных случаях жалуются на некорректную работу компилятора. Однако такие упреки

не всегда заслужены — различия в поведении программы могут возникать даже тогда, когда он работает вполне нормально. И здесь нет никакого противоречия. Дело в том, что все оптимизации написаны с таким расчетом, чтобы переводить одно семантически правильное с точки зрения решения поставленной задачи поведение программы в эквивалентное семантически правильное поведение. Однако в программах (особенно в период их отладки) часто встречаются ошибки. Некоторые из этих ошибок могут быть совершенно безобидными для неоптимизированного кода и не оказывать на него какого-либо существенного влияния. Тем не менее они могут влиять на работу оптимизатора, нарушая часть из негласных предположений, лежащих в основе некоторых из производимых оптимизаций и, таким образом, вызывая появление более серьезных ошибок, непосредственно заметных пользователю.

Другой причиной несоответствия результатов выполнения оптимизированной и неоптимизированной версий программы могут служить конструкции, детали реализации которых не до конца определены спецификацией языка программирования. В качестве наиболее известного примера такой конструкции можно упомянуть инструкцию вызова функции языка C++, в стандарте которого [1] указано, что аргументы функции могут вычисляться в произвольном порядке. В результате если программа написана не вполне корректно, то изменение оптимизатором порядка вычисления аргументов в каком-либо вызове одной из функций может привести к появлению ошибки, которая ранее никак себя не проявляла. Так как обычный отладчик умеет работать только с неоптимизированным кодом программы, то найти причину возникновения такой ошибки будет очень трудно. В то же время если бы при отладке можно было использовать оптимизированный код, то такая ошибка была бы обнаружена программистом при первом же отладочном проходе через соответствующий участок программы.

Автор выражает глубокую признательность Н. Н. Вояковской за множество полезных советов и критических замечаний, позволивших существенно улучшить текст данной статьи.

1. Проблемы, возникающие при отладке оптимизированного кода

Итак, работа с оптимизированным кодом является действительно востребованной функцией, ожидаемой многими пользователями

от отладчика. Но почему же тогда соответствующая поддержка до сих пор не реализована в большинстве промышленных разработок? Ответ на этот вопрос достаточно прост: предоставление подобной возможности является весьма сложной задачей, простого и универсального решения которой пока еще не найдено. При написании такого отладчика его разработчик неминуемо сталкивается с целым рядом проблем, не встречавшихся в случае работы с неоптимизированным кодом, двумя наиболее общими из которых являются *проблема установления местоположения кода* (*code location problem*) и *проблема нахождения значений переменных* (*data value problem*).

Первая проблема возникает из-за того, что установить взаимное соответствие между инструкциями оптимизированной программы и исходным текстом теперь уже намного сложнее, чем в классической схеме отладки. Оптимизатор не только перемещает инструкции исходной программы друг относительно друга, но и заменяет одни инструкции другими, раскручивает циклы, встраивает функции, удаляет мертвый код, а также производит множество аналогичных нетривиальных оптимизаций. Понять, как же соотносятся оптимизированная и исходная программы, после таких изменений бывает сложно даже при ручном, неавтоматическом их сопоставлении.

Однако при реализации отладчика жизненно необходимо уметь строить такое соответствие как в одну, так и в другую сторону. Например, при установке пользователем точки останова на одной из инструкций исходной программы, отладчик должен решить, в каком месте (а иногда даже в каких местах) следует поставить эту точку останова в оптимизированной программе. И наоборот, при возникновении исключительной ситуации в выполняющейся оптимизированной программе, нужно найти соответствующую инструкцию исходной программы и сообщить пользователю о том, что в ней произошла ошибка. В итоге получается, что проблема установления местоположения кода становится одной из основных проблем, которые при построении отладчика, работающего с оптимизированным кодом, приходится решать в первую очередь.

Не менее актуальна и проблема нахождения значений переменных. В оптимизированном коде некоторые переменные исходной программы могут стать неиспользуемыми и попасть в зону действия оптимизации устранения мертвых переменных, а другие, оставшиеся в живых, изменять свои значения совсем не в тех местах, где это происходило в исходной программе.

Например, если в результате оптимизации из цикла была вынесена инструкция, присваивающая некоторое значение одной из переменных, то в оптимизированной программе рассматриваемая переменная изменит свое значение раньше (или позже — в зависимости от того, куда была вынесена инструкция относительно данного цикла), чем это произошло бы в исходной программе. Но так как пользователь при отладке ориентируется исключительно на исходный текст своей программы, то подобное поведение будет выглядеть весьма странным и, скорее всего, только запутает его. Таким образом, если отладчик не будет уделять достаточного внимания проблеме нахождения значений переменных, то его полезность может фактически свестись к нулю.

Однако при разработке отладчика оптимизированного кода приходится одновременно бороться и с целым рядом других трудностей. В частности, необходимо обеспечивать отладку программы в пошаговом режиме (а в оптимизированном коде инструкции могут выполняться совсем в другой последовательности), отображать пользователю текущее состояние стека (что может быть затруднено оптимизациями встраивания функций), заботиться об определении мест хранения значения переменной в каждой точке работы программы (в памяти, на регистрах или в исходном коде в виде константы времени компиляции), предоставлять возможность изменения этого значения, а также решать множество других аналогичных проблем.

2. Способы обеспечения отладки оптимизированного кода

Долгое время указанные проблемы в совокупности образовывали непреодолимый барьер, не позволявший разработать отладчик оптимизированного кода. По мере усложнения алгоритмов оптимизации и увеличения объемов кода, скомпилированного с их использованием, потребность в подобном отладчике проявлялась все более остро. В результате в начале 90-х годов долгожданный прорыв все-таки был осуществлен. В свет вышло сразу несколько работ, сформировавших два основных подхода к построению такого отладчика — *обеспечение ожидаемого поведения* (*providing expected behaviour*) и *обеспечение истинного поведения* (*providing truthful behaviour*).

2.1. Обеспечение ожидаемого поведения

Основная идея подхода, именуемого обеспечением ожидаемого поведения, состоит в том, чтобы полностью скрыть все эффекты оптимизации от пользователя. Иными словами, при работе программа обязана вести себя в точности так, как это предписано ее исходным кодом: во время пошаговой отладки инструкции должны выполняться в том же порядке, в котором они следуют в исходном тексте, все переменные быть доступны и иметь правильные значения, а стек вызовов процедур отображаться ровно в том виде, который он имел бы в этой точке выполнения неоптимизированной программы.

Данного подхода придерживаются авторы работ [7] и [8]. Так как обеспечить соблюдение всех требований, обозначенных выше, основываясь только на выполнении инструкций оптимизированной программы, практически невозможно, обе работы практикуют переключение на время отладки на соответствующий неоптимизированный код, возвращаясь по ее завершению обратно. Такой прием в большинстве случаев не оказывает влияния на скорость работы отлаживаемой программы, так как выигрыш в производительности, полученный при оптимизации программы, при пошаговом выполнении полностью «съедается» эффектом ожидания ответов пользователя. В случае же, если нужно осуществить вызов функции без произведения ее пошаговой отладки («step over»), она вызывается в своем оптимизированном варианте.

В обеих рассматриваемых работах возможность перехода от оптимизированной версии кода к неоптимизированной и обратно обеспечивается только для ограниченного набора точек программы. В [8] такие точки называются *точками прерывания* (*interrupt points*), а в [7] — *точками замещения на стеке* (*On— Stack Replacement Points, OSR Points*). На компилятор при этом накладывается ограничение, состоящее в том, что ему позволяется производить только те оптимизации, которые не оказывают существенного влияния на состояние программы в данных точках. Фактически оптимизацию можно проводить только между точками прерывания. В результате при отборе возможных кандидатов на роль таких точек приходится учитывать, что, с одной стороны, они не могут стоять слишком плотно (иначе будут существенно ограничены возможности оптимизации), а с другой стороны, должны встречаться достаточно регулярно, чтобы отладчик мог перейти к выполне-

нию неоптимизированного кода как можно ближе к тому месту, где пользователь поставил точку останова. Обоим этим требованиям удовлетворяет решение, использованное в работе [8], где в качестве таких точек предлагается рассматривать точки входа в функцию и перехода к следующей итерации цикла. Эти точки, с одной стороны, встречаются практически повсеместно, а с другой стороны, практически не затрагиваются большинством оптимизаций.

Для перехода к неоптимизированному коду [7] и [8] используют две принципиально разные схемы. В работе [8] предлагается осуществлять деоптимизацию только самого верхнего фрейма стека вызовов процедур. В ситуациях, когда требуется деоптимизировать фрейм, находящийся в глубине стека, для него используется *механизм отложенной деоптимизации* (*lazy deoptimization*), то есть выполнение необходимых изменений откладывается вплоть до того момента, когда этот фрейм окажется на вершине стека. Если обнаруживается, что деоптимизируемый фрейм содержит одну или несколько выполняющихся в данный момент *встроенных функций* (*inlined functions*), то для каждой из них на стеке вызовов дополнительно будет создан свой отдельный фрейм. Для того чтобы определить, выполняется ли в данный момент каждая из *встроенных функций*, используется сгенерированное компилятором отображение реального счетчика команд оптимизированной программы на виртуальный счетчик команд исходной.

В работе [7] переход к неоптимизированному коду осуществляется иначе: стек вызовов процедур очищается, и на нем в нужном порядке создаются фреймы неоптимизированных функций. С целью упрощения этой операции для каждого метода, выполнявшегося в момент осуществления перехода, динамически генерируется его вспомогательный вариант. Во вспомогательном варианте этому методу предшествует код, инициализирующий локальные переменные их текущими значениями и загружающий на стек все обрабатываемые данные. Если строящийся фрейм не является последним в стеке вызовов, то в описанный вспомогательный вариант метода добавляется еще и вызов вспомогательного метода следующей процедуры.

Возврат к выполнению оптимизированного кода после завершения отладки тоже реализован в каждой работе по-своему. В [7] для этого используется уже описанная схема, с помощью которой осуществлялся переход к неоптимизированному коду (разработанный алгоритм настолько универсален, что позволяет менять даже уро-

вень оптимизации каждого конкретного метода прямо во время его выполнения). В [8] же приходится дожидаться завершения работы всех деоптимизированных в процессе отладки функций и, таким образом, «естественного» снятия их фреймов со стека (теоретически это может занять достаточно много времени, так как в деоптимизированных функциях могут находиться циклы и вызовы других функций).

Кроме процедуры перехода между оптимизированной и неоптимизированной версией выполняющейся программы, работы данного направления отличаются от обычных отладчиков, имеющих дело с неоптимизированным кодом, незначительно. Действительно, все основные проблемы решаются практически сами собой: проблема установления местоположения кода сводится к хранению информации о наборе точек прерывания данной программы, а проблема нахождения значений переменных и вовсе исчезает, так как при отладке пошаговое выполнение происходит в точном соответствии с исходным текстом, предоставленным пользователем.

Однако у данного подхода есть и свой серьезный недостаток. Дело в том, что фактически в отладке участвует все-таки именно неоптимизированный код. В случае, когда ошибка проявляется только в оптимизированной программе, этот метод может оказаться бесполезным для ее обнаружения.

2.2. Обеспечение истинного поведения

Подобного недостатка лишено направление, называемое обеспечением истинного поведения. Оно основывается на кардинально противоположном принципе — при отладке должен использоваться только реально выполняющийся оптимизированный код, а пользователю показывается только та информация, о которой точно известно, что она отражает состояние исходной программы на данном участке кода. То есть если какие-либо переменные исходной программы недоступны, имеют или могут иметь неправильные значения в данной точке останова, то отладчик при их предоставлении должен честно предупреждать об этом пользователя.

Однако, из-за влияния проблемы установления местоположения кода, уже сам способ задания точки останова в оптимизированной программе нуждается в дополнительной спецификации. Принято различать *семантическое соответствие точек останова* (*se-*

mantic breakpoints mapping) и *синтаксическое соответствие точек останова* (*syntactic breakpoints mapping*).

В первом случае точка останова ставится так, чтобы она была задействована непосредственно перед выполнением той инструкции, на которой она была установлена в неоптимизированной программе. Такая стратегия предоставляет пользователю возможность проанализировать состояние программы до оказания на него какого-либо влияния со стороны этой инструкции. Подобное поведение может потребоваться, например, при установке точки останова на операции присваивания исходной программы, если пользователю нужно узнать значение, хранившееся в переменной непосредственно до выполнения этого присваивания.

В случае же выбора точек останова таким образом, чтобы порядок их выполнения в оптимизированном коде сохранялся, принято говорить о синтаксическом соответствии точек останова. Одним из следствий такого выбора, в частности, является то, что точка останова, поставленная пользователем внутри некоторой конструкции исходной программы, и в оптимизированной программе попадет внутрь соответствующей конструкции. Так, если точка останова была установлена на операции присваивания, вынесенной при оптимизации из цикла, то в оптимизированной программе она все равно будет поставлена на том самом месте внутри этого цикла, откуда исходная инструкция была перемещена.

К сожалению, как семантическое, так и синтаксическое соответствие не всегда задают точку останова однозначно. Основной причиной такой неоднозначности является тот факт, что по большинству инструкций исходной программы компилятор генерирует, как правило, сразу несколько инструкций оптимизированной. Поэтому при практическом применении способ установки точек останова нуждается в дополнительном уточнении.

Примером такого уточнения для семантического соответствия может служить понятие *ключевой инструкции* (*key instruction*), используемое в работе [12]. Ключевой инструкцией некоторой простой команды исходной программы называется инструкция оптимизированного кода, результат выполнения которой непосредственно виден пользователю. Для того чтобы соответствие получилось однозначным, команды исходной программы, имеющие несколько наблюдаемых эффектов (например такие, как циклы со счетчиком), рассматриваются как комбинации более простых, на каждой из которых можно поставить свою точку останова.

Однако основную трудность для работ данного направления представляет все-таки именно решение проблемы нахождения значений переменных. Так, уже только для описания различных состояний переменных вводится большое количество дополнительных терминов. Например, различают *ожидаемое значение переменной* (*expected value*) и *фактическое значение переменной* (*actual value*). Об ожидаемом значении переменной принято говорить при указании на значение, которое она имела бы в данной точке останова при выполнении неоптимизированной программы, в то время как фактическим значением переменной называют значение, которое она реально имеет в данный момент выполнения оптимизированной программы, фактически и выполняющейся при отладке. Заметим, что фактического значения в некоторых точках выполнения программы может и не быть. Такая ситуация наблюдается в случаях, когда переменная либо еще не инициализирована, либо является *нерезидентной* (*nonresident*), т.е. уже замещена в памяти и на регистрах другими переменными. Во всех остальных случаях про переменную в рассматриваемой точке останова можно сказать, что она либо *имеет текущее значение* (*is current*) (если фактическое значение совпадает с ожидаемым значением всегда), либо *имеет сомнительное значение* (*is suspect*) (если фактическое значение в некоторых случаях совпадает с ожидаемым значением, а в некоторых — нет), либо *имеет нетекущее значение* (*is noncurrent*) (когда фактическое и ожидаемое значения в данной точке не совпадают при выборе любого пути выполнения программы).

Практически все алгоритмы, разработанные в рамках метода обеспечения истинного поведения, основаны на анализе графа потока управления. При необходимости проведения более глубокого анализа используются либо граф потоков данных [3, 9], либо подсказки компилятора, как, например, в работах [4, 10], в которых требуется, чтобы компилятором была предоставлена вся необходимая информация о произведенных оптимизациях.

Один из первых алгоритмов данного направления был предложен в работе [5] и доработан впоследствии в [13]. В данном алгоритме используется модифицированный граф потока управления, циклы которого представлены в специальном «раскрученном» виде. Итеративно производя различные операции над группами узлов такого графа, можно за конечное число шагов найти множество всех переменных, имеющих текущие значения в данной точке останова.

Однако анализ только графа потока управления не всегда дает возможность найти все ожидаемые значения переменных, так или иначе хранящиеся в оптимизированной программе. В частности, если согласно произведенным вычислениям переменная на данном участке программы уже не является живой и не имеет представления в памяти, ее значение все еще может храниться на регистрах. Для выявления всех подобных значений в работе [3] применяется анализ графа потоков данных, позволяющий в некоторых случаях достичь двукратного уменьшения количества нерезидентных переменных.

Описанная выше ситуация во многом аналогична рассматриваемой в работе [2]. Здесь исследуется проблема предоставления пользователю значений переменных, вычисление которых в результате применения оптимизаций перемещения кода было произведено раньше ожидаемого момента, либо наоборот будет выполнено позже. В данном случае требуемое значение может быть легко предоставлено, если оно либо уже вычислено, но еще не записано в нужную переменную, либо до сих пор сохранилось на регистрах. Иногда значение может быть предоставлено и в случаях, когда можно найти исходные данные для его вычисления, а вычисляемая операция не очень сложна.

Другим примером, когда отладчик оптимизированного кода может упустить возможность предоставить имеющееся у него ожидаемое значение переменной, является ситуация, когда это значение хранится во временной переменной, либо в другой переменной исходной программы. В случае, когда пользователь запрашивает переменную, ожидаемое значение которой на данный момент недоступно, отладчик, обладая информацией о произведенных оптимизациях, может попробовать найти его среди других имеющихся в его распоряжении переменных. Пример решения подобной проблемы приведен в [4], где описан способ нахождения значения переменной, удаленной в результате устранения неиспользуемых переменных, выполненного после оптимизации распространения копирований.

Немного с другой стороны к предоставлению пользователю уже имеющихся ожидаемых значений запрашиваемой переменной подошли авторы работы [6]. В ней производится динамический анализ текущего (на момент прохождения точки останова) состояния сомнительных переменных. Для этого в отлаживаемую программу при компиляции добавляется вспомогательный код, генерирующий

при выполнении каждого базисного блока специальную временну́ю метку и запоминающий ее для последующей обработки. Если во время отладки запрашивается переменная, имеющая сомнительное значение, то, перед тем как предоставить ее пользователю, по собранным временны́м меткам прохождения базовых блоков строится «раскрученный» граф потока управления, аналогичный разработанному в [13]. Анализируя этот граф, можно определить, совпадает ли в данном конкретном случае фактическое значение переменной с ее жидаемым значением и, соответственно, выдавать или не выдавать пользователю предупреждение об этом.

Для предоставления значений нерезидентных переменных в работе [11] использован метод установки *скрытых точек останова* (*hidden breakpoints*) — точек останова, используемых только отладчиком, существование которых практически незаметно для пользователя. Скрытые точки останова устанавливаются во всех местах, где последнее живое определение переменной в памяти или на регистрах замещается значением другой переменной. При прохождении такой точки выполнение оптимизированного кода приостанавливается, теряемое значение переменной запоминается отладчиком, после чего программе разрешается продолжить свою работу.

Так как нерезидентные переменные могут встречаться достаточно часто (согласно исследованиям, проведенным в [4], при дополнении обычного набора оптимизаций промежуточного представления оптимизациями распределения регистров, в среднем около 50% переменных в каждой точке останова являются нерезидентными), то установка скрытых точек останова для их сохранения существенно замедлила бы работу программы. Поэтому скрытые точки останова размещаются только внутри тех методов, которые уже имеют на данный момент активные пользовательские точки останова. Таким образом предоставляется возможность получать значения всех нерезидентных переменных внутри функций, в которых пользователь, скорее всего, и будет производить отладку (при выходе из этих функций такая возможность в данном случае теряется).

Другой метод восстановления нерезидентных переменных и переменных, имеющих сомнительные или нетекущие значения, применяется в работе [9]. Здесь, используя информацию графа потоков управления/данных исходной программы, сначала определяется, какие значения могут потребоваться для вычисления запрашиваемой пользователем переменной, потом производится их поиск в графе потоков управления/данных оптимизированной программы,

и выполняются необходимые операции, вычисляющие запрашиваемое значение. После этого, если не произошло никаких ошибок, результаты вычислений могут быть предоставлены пользователю.

Кроме проблем установления местоположения кода и нахождения значений переменных, при создании отладчика, основанного на подходе обеспечения истинного поведения, его разработчику приходится сталкиваться с еще одной серьезной проблемой — практической невозможностью изменения значений большинства переменных оптимизированной программы. Ее решению посвящена работа [10], в которой, используя переданную компилятором дополнительную информацию о произведенных над программой преобразованиях, такая возможность предоставляется для ограниченного набора оптимизаций. В частности, поддерживается распространение констант и выполнение константных вычислений, распространение копирования, устранение общих подвыражений и перемещение кода. Однако в некоторых случаях (даже если пренебречь эффектами взаимного влияния результатов оптимизаций друг на друга) изменение значения части переменных все-таки оказывается невозможным.

Таким образом, все работы, основанные на методе обеспечения истинного поведения, в некоторых случаях вынуждены констатировать невозможность предоставления пользователю значений запрашиваемых им переменных. Логично предположить, что круг таких переменных по мере продолжения исследований в данной области постепенно будет сужаться. Однако обеспечить предоставление пользователю абсолютно всех ожидаемых значений переменных, доступных в исходной программе, без существенного вмешательства в работу оптимизированной программы, скорее всего, невозможно.

Тем не менее в некоторых случаях предоставляемых значений может оказаться достаточно для решения возложенных на отладчик задач, а возможность отладки неизмененного оптимизированного кода может с лихвой компенсировать все причиняемые неудобства.

Заключение

Сравнивая приведенные методы отладки оптимизированного кода, приходится констатировать, что ни один из них не справляется с поставленной задачей в полной мере. Так, хотя подход, осно-

ванный на обеспечении ожидаемого поведения, и позволяет выполнять весь спектр стандартных отладочных операций при практически полном отсутствии ограничений, накладываемых на работу оптимизатора, но из-за того, что непосредственно во время отладки выполняется все-таки неоптимизированный код, практическая применимость данного метода оказывается намного ниже, чем того хотелось бы.

Но и метод обеспечения истинного поведения не предоставляет пользователю идеальной свободы. Хотя в нем отлаживается как раз именно оптимизированный код, невозможность получения значений какой-либо части переменных может свести на нет все приложенные к отладке усилия. Однако при ограниченном наборе оптимизаций он, скорее всего, вполне может оказаться применимым на практике.

Список литературы

[1] *Страуструп Б.* Язык программирования C++. — СПб.: Невский Диалект. — 1999. — 991 с.

[2] *Adl-Tabatabai A., Gross T.* Detection and Recovery of Endangered Variables Caused by Instruction Scheduling // ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. — Vol. 28. — Issue 6. — 1993. — P. 13–25.

[3] *Adl-Tabatabai A., Gross T.* Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging // 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — 1993. — P. 371–383.

[4] *Adl-Tabatabai A., Gross T.* Source-level Debugging of Scalar Optimized Code // ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation. — Vol. 31. — Issue 5. — 1996. — P. 33–43.

[5] *Berger L., Wismüller R.* Source-level Debugging of Optimized Programs Using Data Flow Analysis. — http://citeseer.ist.psu.edu/130648.html. — 1993. — 21 p.

[6] *Dhamdhere D. M., Sankaranarayanan K. V.* Dynamic Currency Determination in Optimized Programs // ACM Transactions on Programming Languages and Systems. — Vol. 20. — Issue 6. — 1998. — P. 1111–1130.

[7] *Fink S. J., Qian F.* Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement // IBM T.J. Watson Research Center. — 2003. — 12 p.

[8] *Hölzle U., Chambers C., Ungar D.* Debugging Optimized Code with

Dynamic Deoptimization // ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. — Vol. 27. — Issue 7. — 1992. — P. 32–43.

[9] *Hong I., Kirovski D., Potkonjak M., Papaefthymiou M. C.* Symbolic Debugging of Globally Optimized Behavioral Specifications. — http://citeseer.ist.psu.edu/547763.html. — 2000. — 4 p.

[10] *Kumar M. S.* Debugging Optimized Code: Value Change Problem. Master of Technology Degree Thesis. — Kanpur, 2001. — 70 p.

[11] *Tice C., Graham S. L.* A Practical Approach for Recovery of Evicted Variables. Research Report 167. — Compaq Systems Research Center. — 2000. — 20 p.

[12] *Tice C., Graham S. L.* Key Instructions: Solving the Code Location Problem for Optimized Code. Research Report 164. — Compaq Systems Research Center. — 2000. — 25 p.

[13] *Wismüller R.* Debugging of Globally Optimized Programs Using Data Flow Analysis // ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation. — Vol. 29. — Issue 6. — 1994. — P. 278–289.