УДК 004.4'422
DOI 10.25205/1818-7900-2018-16-2-78-85

А. Е. Малых

Новосибирский государственный университет
ул. Пирогова, 1, Новосибирск, 630090, Россия

ООО НЦИТ «УНИПРО»
ул. Ляпунова, 2, Новосибирск, 630090, Россия

awa149@rambler.ru

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ РАЗРЕШЕНИЯ КОНФЛИКТОВ ПО ДОСТУПУ К ПАМЯТИ В ДИНАМИЧЕСКОМ КОМПИЛЯТОРЕ JAVA ДЛЯ ПРОЦЕССОРА «ЭЛЬБРУС»

Рассмотрены особенности разработки алгоритмов разрешения конфликтов по доступу к памяти и их реализация в динамическом компиляторе Java для отечественной платформы «Эльбрус». Эти алгоритмы позволяют существенно расширить возможности планировщика инструкций – ключевой оптимизации VLIW-процессоров. В работе исследуются статические и динамические подходы к анализу зависимостей по памяти, и приводится сравнение эффективности реализованных алгоритмов на основе стандартной тестовой сюиты SpecJVM2008.

Ключевые слова: Эльбрус, Java, JIT-компилятор, планировщик инструкций, оптимизации компилятора.

Постановка задачи

В настоящее время идет активное развитие Российской электроники, в том числе и процессоров. Так, процессоры «Эльбрус», разработанные на основе архитектуры Very Long Instruction Word (VLIW), призваны заменить зарубежные аналоги в отраслях, где ключевым критерием является безопасность, – например, в работе государственных организаций.

Одна из важных задач, которая всегда встает перед разработчиками нового процессора, – это задача адаптации уже разработанного программного обеспечения под новый процессор. Для ее решения необходимо разработать для этого процессора компиляторы и виртуальные машины популярных языков программирования. Одним из примеров таких языков является Java – кросс-платформенный язык, в котором программы преобразуются в архитектурно независимый байт-код, а затем исполняются при помощи виртуальной Java-машины (JVM). Идея работы JVM заключается в том, чтобы интерпретировать методы, которые используются в программе не очень часто, и компилировать в машинный код часто встречающиеся методы. При этом компилятор должен обеспечивать качество кода с помощью встроенных методов оптимизации.

Как и на любом процессоре с архитектурой VLIW, важной оптимизирующей компонентой компилятора является планировщик инструкций. В текущей версии компилятора Java для процессора «Эльбрус» используется суперблоковый планировщик инструкций [1], позволяющий выбрать сразу несколько базовых блоков и запланировать операции внутри них так, как если бы планирование происходило в одном блоке. Зачастую при планировании возникает ситуация, когда одна инструкция должна быть исполнена раньше другой, в таких случаях речь идет о *зависимости* между инструкциями. Если не получается точно определить, есть

Малых А. Е. Разработка и реализация алгоритмов разрешения конфликтов по доступу к памяти в динамическом компиляторе Java для процессора «Эльбрус» // Вестн. НГУ. Серия: Информационные технологии. 2018. Т. 16, № 2. С. 78–85.

ли зависимость между какой-либо парой инструкций, то необходимо предполагать ее наличие, чтобы не допустить ошибку выполнения программы. Одним из видов зависимости является *зависимость по памяти*, когда операции «загрузка-запись» и «запись-запись» могут указывать на один и тот же участок памяти в программе. В предыдущей реализации JVM для процессора «Эльбрус» любая пара вышеупомянутых обращений к памяти считалась зависимой, что отрицательно сказывалось на возможностях планирования [2]. Целью данной работы являлся анализ алгоритмов разрешения конфликтов по доступу к памяти и их реализация в JVM на процессоре «Эльбрус».

В работе анализируются существующие в настоящее время алгоритмы и способы их адаптации к JVM, а также описываются новые алгоритмы, разработанные с учетом особенностей JVM и процессора «Эльбрус». В заключение приводится сравнение производительности реализованных алгоритмов.

Обзор предметной области

Виртуальная Java-машина. Java – это типизированный объектно-ориентированный язык, разработанный компанией Sun Microsystems. Исходные коды на Java транслируются в .class-файлы, содержащие специальный байт-код, благодаря которому обеспечивается кросс-платформенность, – такие файлы могут быть исполнены на любой архитектуре, для которой реализована JVM.

Для исполнения байт-кода JVM на процессоре «Эльбрус» включает в себя интерпретатор и JIT-компилятор. Интерпретатор исполняет один байт-код за другим, не применяя при этом сложных оптимизаций, он используется для реализации методов, исполняющихся не очень часто. JIT-компилятор, в свою очередь, анализирует и оптимизирует сразу весь метод, в результате чего получается более оптимальный код. Так как сам процесс компиляции является ресурсоемким, он обычно применяется к часто исполняющимся методам [3].

Аппаратные возможности процессора «Эльбрус». Ключевая особенность процессоров «Эльбрус» состоит в том, что они построены на VLIW-архитектуре, позволяющей за один такт процессора выполнять сразу несколько инструкций. Эти инструкции исполняются параллельно, а распределение работ между ними задается во время компиляции. Такой подход существенно упрощает устройство процессора и позволяет увеличить количество вычислительных модулей, однако усложняет работу компилятора [4].

За один такт процессор выполняет одну *широкую команду*. Такая команда может содержать несколько операций сложения, вычитания, умножения, деления, операций загрузки и записи и др.

Планировщик инструкций. Задача планирования заключается в том, чтобы выбрать последовательность выполнения инструкций, минимизировав при этом сумму времени исполнения операций и времени простоя [5]. Для ее решения нужно разместить все инструкции в наименьшее количество широких команд, соблюдая при этом минимальные задержки по готовности результатов.

Ключевым понятием при планировании является определение *базового блока*. Базовый блок – это последовательность инструкций, имеющая одну точку входа, одну точку выхода и не содержащая инструкций передачи управления ранее точки выхода [6].

Один из вариантов реализации планировщика заключается в том, чтобы производить планирование инструкций только внутри базовых блоков. Проблема такого подхода заключается в том, что зачастую код содержит небольшие блоки, внутри которых недостаточное количество инструкций, для того чтобы выполнить эффективную упаковку кода.

Один из способов решения этой проблемы заключается в том, чтобы производить *суперблоковое планирование*. *Суперблок* – это последовательность базовых блоков, содержащая только одну точку входа и сколько угодно точек выхода [7]. Используемый в JIT-компиляторе суперблоковый планировщик был разработан с учетом особенностей JVM для процессора «Эльбрус» [1].

Суперблоковый планировщик – это итеративный алгоритм, каждая итерация которого выглядит следующим образом:

- выбрать еще не запланированный суперблок;
- построить граф зависимостей между инструкциями в суперблоке;

- выбрать порядок выполнения инструкций и разместить их по широким командам.

Построение графа зависимостей. Одним из ключевых понятий алгоритмов планирования инструкций является граф зависимостей – это взвешенный ориентированный граф, описывающий зависимости между инструкциями [8]. Множество вершин такого графа совпадает со множеством инструкций, для которых производится планирование. Ребро (u, v) с весом w в графе указывает, что инструкция u должна быть исполнена раньше инструкции v хотя бы на w тактов. Если w равно 0, значит, инструкции u и v могут быть исполнены в одной широкой команде. Тем не менее надо учитывать, что в некоторых случаях порядок инструкций в широкой команде важен, и инструкция u все равно должна быть исполнена раньше инструкции v.

Можно выделить три ключевых вида зависимостей между инструкциями:

- *по данным* (возникают, когда операции чтения и записи значения в регистр должны следовать друг за другом в определенном порядке);
- *по памяти* (возникают, когда операции загрузки и записи в память должны следовать друг за другом в определенном порядке);
- *по управлению* (возникают, когда необходимо выполнить какую-то инструкцию гарантированно до или после перехода в другой блок).

Чем меньше ребер окажется в итоговом графе зависимостей, тем больше будет в результате возможностей для планирования.

Зависимости по данным определяются однозначно и не могут быть упрощены. Количество зависимостей по управлению может быть уменьшено путем вставки дополнительного кода, восстанавливающего состояния регистров на выходах из суперблока. На данный момент эта оптимизация уже реализована в JVM для «Эльбруса».

Таким образом, с точки зрения уменьшения количества зависимостей наибольший интерес представляют зависимости по памяти. Если удастся доказать, что две инструкции обращаются к разным участкам памяти, то можно говорить о том, что они независимы [9].

Алгоритмы разрешения конфликтов по доступу к памяти

Рассмотрим два основных подхода к анализу зависимостей по доступу к памяти: *статический* и *динамический*. Статический подход заключается в том, чтобы производить проверку зависимости двух операций обращения к памяти во время компиляции метода, и в соответствии с результатом проверки либо добавлять ребро в графе зависимостей, либо не добавлять. Идея динамического подхода, в свою очередь, заключается в том, чтобы производить такую проверку во время исполнения метода. В таком случае нам необходимо генерировать код для обоих исходов проверки [10].

Статический анализ по типам. Одной из ключевых особенностей языка Java является отсутствие указателей, благодаря чему представляется возможным выполнить анализ по типам – статический подход к анализу зависимостей по памяти при помощи сравнения типов загружаемых и записываемых элементов [11].

В высокоуровневом представлении компилятора хранится дополнительная информация об объектах, в частности о классах, к которым эти объекты относятся. Здесь важно отметить, что из-за наличия в языке Java виртуального полиморфизма на этапе компиляции мы не можем достоверно знать, принадлежит в действительности объект к данном классу или к одному из его наследников.

Передав информацию о классах объектов в низкоуровневое представление, мы получаем возможность определять, к каким классам относятся загружаемые и записываемые элементы на этапе построения графа зависимостей (Здесь неважно, является эта операция обращением к полю некоторого объекта или обращением к индексу массива. В первом случае речь будет идти о классе, к полю которого мы обращаемся, во втором – о классе массива.) Если класс объекта, из которого идет загрузка, является наследником или предком класса объекта, в который производится запись, то возможно, что эти объекты совпадают, и соответствующие операции могут указывать на один и тот же адрес. В этом случае нам необходимо использовать другие методы анализа зависимостей. Если же это не так, т. е. ни один из этих классов

не является наследником другого, то можно утверждать, что операции обращения к памяти являются независимыми, и не добавлять между ними ребро в графе зависимостей.

Также анализ особенностей языка Java показал, что в результате наследования классов типы полей объектов не могут быть изменены. Тем самым мы можем анализировать по типам операции чтения и записи не только на основе объектов, из которых происходит загрузка, но и на основе типов непосредственно загружаемых объектов (например, классов полей при обращении к полю объекта). В этом случае нам даже не нужно смотреть на иерархию классов, достаточно просто сравнивать, совпадает ли класс загружаемого объекта с классом записываемого. Если классы отличаются между собой, то операции гарантированно являются независимыми.

Динамический анализ внутри циклов. Самый простой способ сгенерировать динамическую проверку – это добавить блок, в котором производится сравнение на равенство двух адресов чтения и записи, и создать код для обоих возможных случаев выполнения программы – один для случая равенства адресов и другой для случая неравенства. При таком подходе надо учитывать, что проверка будет производиться во время исполнения программы, поэтому генерировать такие проверки для всех пар операций слишком ресурсоемко. Обычно проверки создаются только для наиболее «горячих» участков кода, а именно – для циклов. Если адреса в операциях обращения к памяти не зависят от номера итерации цикла, то можно сделать блок с проверкой перед циклом и создать два варианта цикла – оптимизированный, в котором все операции чтения и записи между собой не пересекаются, и неоптимизированный, в котором операции чтения и записи считаются пересекающимися, если обратное не было доказано при помощи статических алгоритмов разрешения конфликтов по доступу к памяти. Если в цикле есть несколько операций чтения и записи, то нужно добавить проверку для всех таких пар и делать переход на оптимизированную версию цикла только в случае, если никакие из них не являются пересекающимися.

Как правило, в реальных программах адреса операций чтения и записи в цикле зависят от номера итерации, поэтому для эффективной работы динамической проверки нужен более сложный анализ, для которого необходимо, чтобы цикл имел канонический вид, а именно:

- содержал ровно одну индуктивную переменную;
- индуктивная переменная изменялась на каждом шаге на некоторую константу, известную на этапе компиляции;
- не имел других точек входа кроме головы цикла;
- не имел других точек выхода кроме хвоста цикла.

В таких циклах инструкции обращения к памяти в массивах можно представить в виде

$$адрес = база + множитель \times индуктивная\ переменная + смещение.$$

Для получения такого представления необходимо определить, каким образом получается адрес обращения к памяти. Для этого мы находим предыдущую операцию, в которой происходило определение регистра, соответствующего адресу, и в зависимости от типа операции изменяем базу, множитель и смещение. Далее процедура повторяется для базы, пока мы не дойдем до операции, из которой нельзя будет однозначно выразить новые значения базы и смещения (это может быть, например, загрузка из памяти или результат вызова функции). Когда такое представление будет получено для каждой инструкции обращения к памяти в цикле, мы можем проводить статический анализ для соответствующих операций, а именно: если для каких-то двух операций базы и множители совпадают, а смещения различаются, то мы можем утверждать, что они указывают на разную память. Несовершенство такого подхода заключается в том, что часто базы, с которых происходит загрузка, различаются (например, два массива передаются в функцию в качестве аргументов), и в этом случае мы ничего не можем сказать о зависимости между ними. Для преодоления этой проблемы перед циклом выполняется динамическая проверка. Мы составляем список используемых в цикле баз массивов, а перед циклом вставляем дополнительный проверочный блок. В этом блоке сравниваются между собой базы используемых в цикле массивов, и если базы всех массивов отличаются друг от друга, то делается переход на предварительно сгенерированную оптимизированную версию цикла. В противном случае делается переход на неоптимизированную

версию [12]. Преимущество описанного подхода состоит в том, что при планировании нашей оптимизированной версии цикла мы можем полагать, что никакие базы между собой не пересекаются, и, соответственно, если адреса двух инструкций определяются через две разных базы, то они гарантированно являются независимыми.

Динамический анализ с использованием аппаратных возможностей процессора «Эльбрус». Архитектура процессора «Эльбрус» предоставляет специальное устройство для динамического разрешения зависимостей по памяти. Принцип работы устройства основывается на разбиении начальной инструкции чтения на две: *preload* и *check*. Обе инструкции содержат три аргумента, два из которых определяют адрес загрузки, а третий обозначает регистр, в который будет записан результат чтения. Инструкция *preload* выполняет обычную загрузку из памяти и добавляет адрес, с которого произошла загрузка, в специальную таблицу процессора. Любая последующая операция записи с этого адреса удаляет соответствующую строчку из таблицы. Инструкция *check* проверяет наличие данного адреса в таблице. Если адрес в таблице есть, то значение загруженного ранее регистра актуально, и указанный в качестве аргумента адрес просто удаляется из таблицы. Если же такого адреса в таблице нет, значит, данные по нему были перезаписаны, в таком случае загрузка производится заново, как при обычной операции чтения. Кроме того, в последнем случае выставляется специальный флаг, по которому впоследствии это можно сделать. Благодаря этому мы можем выносить за инструкции записи не только загрузку, но и зависимые от нее операции, и если эти зависимые инструкции были произведены с «неправильным» значением загруженного регистра, то мы можем исполнить их снова. Такие зависимые операции называются *компенсационным кодом*.

Для эффективного использования соответствующих команд «Эльбруса» необходимо внести определенные изменения в вышеприведенный алгоритм суперблокового планирования. Модифицированная версия планировщика выглядит следующим образом [13]:

- выбрать еще не запланированный суперблок;
- разбить каждую инструкцию чтения на *preload* и *check*;
- построить граф зависимостей, полагая инструкции *preload* независимыми по отношению к инструкциям записи;
- спланировать инструкции в выбранном суперблоке, убрав лишние инструкции *check*;
- добавить компенсационный код.

Несмотря на то что идея алгоритма достаточно простая, есть несколько важных моментов, которые нужно учитывать при реализации алгоритма. Особенность инструкции *check* на «Эльбрусе» заключается в том, что она не может быть спекулятивной, а значит, в результате должна остаться в исходном блоке, что необходимо отдельно учитывать при построении графа зависимостей. Кроме того, в JVM на «Эльбрусе» используются *неявные исключения*, т. е. может быть сделана загрузка с нулевого адреса и только потом произведена проверка на то, удалось ли сделать такую загрузку. Чтобы избежать загрузки с нулевого адреса, неявная проверка переносится между инструкциями *preload* и *check*. Также определяемый в инструкции *preload* регистр добавляется в *check* в качестве дополнительного аргумента, чтобы при распределении регистров между ними на то же место не был назначен другой регистр.

Была разработана эвристика, которая не дает выносить инструкции *preload* слишком высоко от начального места, это позволяет уменьшить количество срабатываний инструкции *check* и сократить давление на регистры, чтобы избежать ситуации, когда нам приходится выгружать часть регистрового файла на стек.

Чтобы корректно сгенерировать компенсационный код, для каждой операции *preload* необходимо поддерживать набор регистров, значения которых от нее зависят. Важно заметить, что нельзя переносить зависимости от использования к определению регистра за инструкцию *check*, так как в этом случае не получится восстановить начальные значения регистров для того, чтобы исполнить их снова.

Результаты

Алгоритмы, описанные в этой статье, были реализованы в динамическом компиляторе с языка Java для VLIW-процессора «Эльбрус». Компилятор с реализованными оптимизациями был проверен на следующих стандартных тестах:

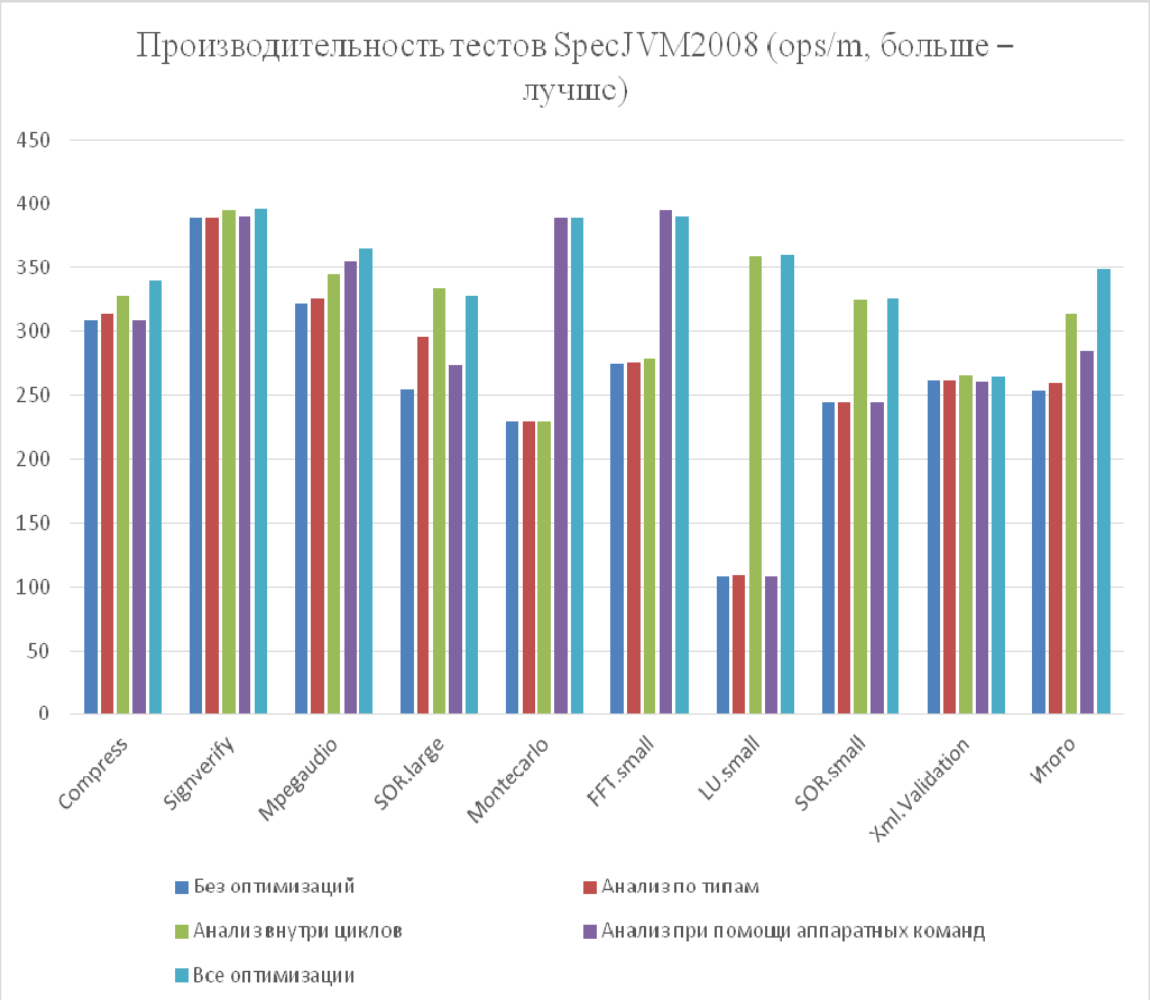
- SpecJVM98;
- SpecJVM2008;
- SpecJBB2005;
- Dacapo;
- JCK;
- JCTF.

Все тесты были пройдены успешно.

Также была произведена оценка полученного ускорения за счет реализованных оптимизаций на стандартной тестовой сюите SpecJVM2008. Тесты SpecJVM2008 включают в себя следующие бенчмарки:

- Compress – алгоритм сжатия LZW;
- Signverify – алгоритм подписи и проверки на основе протоколов MD5withRSA, SHA1withRSA, SHA1withDSA и SHA256withRSA;
- Mpegaudio – декодирование формата mp3;
- Scimrak SOR – метод релаксации Якоби;
- Scimrak Monte Carlo – интегрирование методом Монте-Карло;
- Scimrak LU – LU-разложение матрицы;
- Xml.Validation – валидация xml-документов по xml-схеме.

Следует заметить, что тесты Scimark (кроме Monte Carlo) в SpecJVM2008 включены в small и large версии. В первом случае все данные, которыми оперирует программа, помещаются в кеш процессора, во втором – размеры матриц и массивов подобраны таким образом, что их необходимо загружать из оперативной памяти. Результаты сравнения производительности приведены на рисунке:



Заключение

Статья является логическим продолжением работы [1] над улучшением показателей упаковки кода и, как следствие, ускорением работы JIT-компилятора Java на процессоре «Эльбрус». Разработанные и описанные в статье оптимизации открывают большие возможности для добавления ряда других важных улучшений компилятора. В частности, это конвейеризация циклов и динамическая проверка существования исключений в цикле, которые невозможны без реализации рассмотренных выше алгоритмов. Кроме того, планируется перенести описанные алгоритмы и на другие JIT-компиляторы для процессора «Эльбрус», а именно, C# и JavaScript.

Список литературы

1. *Андреенко С. А.* Межблоковое планирование инструкций в динамическом компиляторе java для VLIW-процессора. Новосибирск, 2016.

2. *Ghosh, Soumyadeep, Yongjun Park, and Arun Raman.* Enabling efficient alias speculation // ACM SIGPLAN Notices. 2015. Vol. 50. No. 5.

3. *Paleczny M., Vick C., Click C.* The Java HotSpot Server Compiler, Sun Microsystems // Java Virtual Machine Researchand Technology Symposium (JVM '01), 2001.

4. Микросхема интегральная 1891ВМ7Я (Система команд): Руководство программиста. ТВГИ.00742-01 33 01-1. Ч. 1. Общие сведения.

5. *Hwu, Wen-Mei W. et al.* The superblock: an effective technique for VLIW and superscalar compilation // Instruction-Level Parallelism. Springer, Boston, MA, 1993. P. 229–248.

6. *Mahlke, Scott A. et al.* Effective compiler support for predicated execution using the hyperblock // ACM SIGMICRO Newsletter. IEEE Computer Society Press, 1992. Vol. 23. No. 1–2.

7. *Faraboschi P., Fisher J. A., Young C.* Instruction scheduling for instruction level parallel processors // Proceedings of the IEEE. 2001. Vol. 89.11. P. 1638–1659.

8. *Yang T., Gerasoulis A.* List scheduling with and without communication delays // Parallel Computing. 1993. Vol. 19.12. P. 1321–1344.

9. *Huang A. S., Slavenburg G., Shen J. P.* Speculative disambiguation: A compilation technique for dynamic memory disambiguation // ACM SIGARCH Computer Architecture News. IEEE Computer Society Press, 1994. Vol. 22. No. 2.

10. *Maalej M. et al.* Pointer disambiguation via strict inequalities // Proceedings of the 2017 International Symposium on Code Generation and Optimization. IEEE Press, 2017.

11. Shpeisman T., Lueh G.-Y., Adl-Tabatabai A-R. Just-in-time Java compilation for the Itanium/spl reg/processor // Parallel Architectures and Compilation Techniques. Proceedings International Conference. IEEE, 2002.

12. *Péricles A. et al.* Runtime pointer disambiguation // ACM SIGPLAN Notices. 2015. Vol. 50. No. 10.

13. *Gallagher D., Chen W., Mahlke S., Gyllenhaal J., Hwu W.* Dynamic Memory Disambiguation Using the Memory ConflictBuffer // ASPLOS-VI Proceedings. Center for Reliable and High-Performance Computing, University of Illinois, 1994.

Материал поступил в редколлегию 17.03.2018

A. E. Malykh

Novosibirsk State University
1 Pirogov Str., Novosibirsk, 630090, Russian Federation

UNIPRO Ltd.
2 Lyapunov Str., Novosibirsk, 630090, Russian Federation

awa149@rambler.ru

DEVELOPMENT AND IMPLEMENTATION OF MEMORY DISAMBIGUATION ALGORITHMS IN DYNAMIC JAVA COMPILER FOR ELBRUS PROCESSOR

This article describes algorithms for memory disambiguation in dynamic Java compiler for Russian processor Elbrus with their implementation. These algorithms let significantly improve possibilities for instruction scheduler which is a key optimization for VLIW-processors. Static and dynamic approaches for memory disambiguation are described in this work. All implemented algorithms' efficiency is shown based on popular testing suite SpecJVM2008.

Keywords: Elbrus, Java, JIT-compiler, instruction scheduler, compiler optimizations.

References

1. Andreenko S. A. Superblock instruction scheduling in dynamic java compiler for VLIW processor. Novosibirsk, 2016. (in Russ.)

2. Ghosh, Soumyadeep, Yongjun Park, and Arun Raman. Enabling efficient alias speculation. *ACM SIGPLAN Notices*, 2015, vol. 50, no. 5.

3. Paleczny M., Vick C., Click C. The Java HotSpot Server Compiler, Sun Microsystems. *Java Virtual Machine Researchand Technology Symposium (JVM '01)*, 2001.

4. Integrated circuit 1891VM7Y, Programmers manual, (Instruction set), TVGI.00742-01 33 01-1. Part 1. General information. (in Russ.)

5. Hwu, Wen-Mei W. et al. The superblock: an effective technique for VLIW and superscalar compilation. *Instruction-Level Parallelism*. Springer, Boston, MA, 1993, p. 229–248.

6. Mahlke, Scott A. et al. Effective compiler support for predicated execution using the hyperblock. *ACM SIGMICRO Newsletter*. IEEE Computer Society Press, 1992, vol. 23, no. 1–2.

7. Faraboschi P., Fisher J. A., Young C. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 2001, vol. 89.11, p. 1638–1659.

8. Yang T., Gerasoulis A. List scheduling with and without communication delays. *Parallel Computing*, 1993, vol. 19.12, p. 1321–1344.

9. Huang A. S., Slavenburg G., Shen J. P. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. *ACM SIGARCH Computer Architecture News*. IEEE Computer Society Press, 1994, vol. 22, no. 2.

10. Maalej M. et al. Pointer disambiguation via strict inequalities. *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017.

11. Shpeisman T., Lueh G.-Y., Adl-Tabatabai A-R. Just-in-time Java compilation for the Itanium/spl reg/processor. *Parallel Architectures and Compilation Techniques. Proceedings International Conference*. IEEE, 2002.

12. Péricles A. et al. Runtime pointer disambiguation. *ACM SIGPLAN Notices*, 2015, vol. 50, no. 10.

13. Gallagher D., Chen W., Mahlke S., Gyllenhaal J., Hwu W. Dynamic Memory Disambiguation Using the Memory ConflictBuffer. *ASPLOS-VI Proceedings*. Center for Reliable and High-Performance Computing, University of Illinois, 1994.

For citation:
Malykh A. E. Development and Implementation of Memory Disambiguation Algorithms in Dynamic Java Compiler for Elbrus Processor. *Vestnik NSU. Series*: *Information Technologies*, 2018, vol. 16, no. 2, p. 78–85. (in Russ.)

DOI 10.25205/1818-7900-2018-16-2-78-85