УДК 004.4’242

ПРИМЕНЕНИЕ МЕТОДА СПЕКУЛЯТИВНОГО ВЫПОЛНЕНИЯ ДЛЯ РАСПАРАЛЛЕЛИВАНИЯ ИСХОДНОГО КОДА, СОДЕРЖАЩЕГО ОБРАБОТКУ ИСКЛЮЧЕНИЙ

Т.Н. Романова, **А.В. Сидорин**

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация
e-mail: rtn@bmstu.ru; alexey.v.sidorin@ya.ru

Рассмотрена проблема распараллеливания программного кода, содержащего обработку исключительных ситуаций. Исследована возможность использования метода спекулятивного выполнения для распараллеливания фрагмента кода, который может сгенерировать исключение. Описаны разработанные алгоритмы для распараллеливания такого рода программных кодов. Приведены результаты тестовых расчетов по разработанной программе на языке Java, подтверждающие перспективность выбранного подхода для решения указанной проблемы.

***Ключевые слова*:** распараллеливание, обработка исключений, спекулятивное выполнение.

APPLICATION OF METHOD OF SPECULATIVE EXECUTION FOR PARALLELIZATION OF SOURCE CODE CONTAINING EXCEPTIONS HANDLING

T.N. Romanova, **A.V. Sidorin**

Bauman Moscow State Technical University, Moscow, Russian Federation
e-mail: rtn@bmstu.ru; alexey.v.sidorin@ya.ru

This article analyzes problem of parallelization of source code containing handling of exceptional situations. We have investigated an application possibility of method of speculating execution in order to parallelize code fragments that might generate exceptions. Developed algorithms have been described for parallelizing such source codes. We present results of test calculations using the program developed in the Java language that confirm prospects of the approach chosen to solve this problem.

***Keywords*:** parallelization, exception handling, speculative execution.

В связи с распространением многопроцессорных конфигураций вычислительных систем в последнее время становится актуальной проблема эффективного применения ресурсов таких систем. Создание программы, способной использовать преимущества многопоточности, требует высокой квалификации как программиста, так и проектировщика системы. Поэтому может оказаться эффективным подход, при котором сначала корректно реализуется однопоточный алгоритм, а затем с помощью автоматических средств проводится преобразование реализованного алгоритма в его многопоточную версию.

Автоматическое распараллеливание программ как на уровне объектного, или байт-кода, при компиляции [1–3], так и на уровне исходного кода (source-to-source) [4] и аппаратном уровне [5] являлось объектом исследований в течение долгого времени и в настоящее время остается в центре внимания. Существуют известные продукты, способные проводить распараллеливание на уровне исходного кода для различных языков, например Intel Parallel Studio. Однако существующие инструменты предъявляют требования к исходному коду приложения (отсутствие определенных инструкций и др.) Таким образом, при распараллеливании кода, не написанного специально для распараллеливающей трансляции, возникают проблемы, одной из которых является обработка исключений.

Многие операции как в составе стандартных библиотек языков, так и в пользовательском исходном коде, используют механизм исключений для оповещения вызывающих функций о некорректных ситуациях, которые могут возникнуть при выполнении указанных операций. Одни языки (например, Java) имеют строгие спецификации исключений, при нарушении которых код не может быть скомпилирован [6]. В таких языках функция или метод, генерирующие (“выбросить”) исключение, должны специфицировать типы выбрасываемых исключений в декларации. Прочие типы исключений, которые могут быть выброшены инструкциями, входящими в тело функции или метода, должны перехватываться в методе или функции. Другие языки (например, C++) относятся к декларации исключений менее строго и позволяют не декларировать типы выбрасываемых исключений, а также указывать типы исключений, которые не выбрасываются функцией или методом [7, 8]. Тем не менее, хотя в некоторых языках нельзя полностью полагаться на декларацию исключения, при распараллеливании исходного кода исключения также обязательно должны быть учтены. Особенно это утверждение актуально для компонентов программ, обеспечивающих ввод-вывод, работу с дисками и периферийным оборудованием.

Суть выброса исключения как операции заключается в том, что в результате генерации исключения стек вызовов раскручивается до тех пор, пока не будет определен стековый кадр функции, в которой находится обработчик этого типа исключений. С такой точки зрения генерацию исключения можно рассматривать как переход вверх по последовательности вызовов. Нередко обработка исключений реализуется именно с помощью безусловных переходов, например механизмом setjmp/longjmp [8, 9]. В связи с этим при распараллеливании кода, содержащего обработку или генерацию исключений, возникают сложности: наличие перехода, ведущего вверх по стеку вызовов (как правило, за границы распараллеливаемого фрагмента кода), в после-

довательном коде мешает применению стандартных практик распараллеливания [10].

Во многих языках обработка исключений — стандартная практика, без применения которой невозможно написать корректно работающую программу. Инструкции для обработки исключений можно часто встретить в пользовательских программах. Поэтому в настоящей работе исследованы способы распараллеливания фрагментов кода, содержащих обработку исключений.

Постановка задачи. Примем обозначения операторов, отвечающих за работу с исключениями, в соответствии со стандартом языка Java [6]:

1) оператор генерации исключения throw;
2) try-блок, содержащий инструкции, которые могут сгенерировать исключения. Исключения, произошедшие в try-блоке, могут быть перехвачены и обработаны. Исключения, произошедшие вне try-блока и внутри его, но не перехваченные ни одним из catch-блоков, переходят на уровень вверх;
3) catch-блок, содержащий обработчик исключения для соответствующего ему try-блока;
4) finally-блок, выполняемый независимо от того, произошло ли исключение в try-блоке.

Следовательно, проблемы при распараллеливании будут наблюдаться в тех случаях, когда фрагмент кода, который может быть выделен в отдельный поток, сгенерирует исключение, обработчик которого находится вне этого фрагмента. Если не принять никаких дополнительных мер, то поток, сгенерировавший исключение, просто аварийно завершится. Потоки, которые выполняются параллельно с потоком, сгенерировавшим исключение, продолжат выполнение, игнорируя отсутствие действий со стороны завершившегося потока. Это может привести программу в состояние, при котором она не эквивалентна своей однопоточной версии, поскольку выходное состояние распараллеленного фрагмента после его выполнения будет отличаться от того, которое имелось бы на выходе исходного фрагмента. Если поток, сгенерировавший исключение, содержит инструкции блокировки (например, при доступе к разделяемым данным потоков), то генерация исключения без освобождения объекта блокировки может привести к взаимной блокировке оставшихся потоков, продолживших выполнение.

Для решения этой проблемы определим следующие требования к результату распараллеливания кода, содержащего исключения.

1. Исключения должны быть сгенерированы в том порядке, в котором они бы были выброшены при использовании однопоточной модели. Допустим, что есть однозначное соответствие между операторами

throw до и после распараллеливания. Тогда, если разные операторы throw сгенерировали исключения в различных потоках, то программа должна повести себя так, как если бы исключение было сгенерировано в операторе, выполняющемся раньше в однопоточной программе.

2. Изменение состояния программы во фрагменте кода, генерирующем исключение, должно соответствовать изменению состояния однопоточного фрагмента кода. Например, если потоки в распараллеленном цикле осуществляют запись в одну и ту же общую переменную, то после генерации исключения значение этой переменной должно соответствовать значению переменной на той итерации однопоточного цикла, где произошло исключение.

Требование 2 можно смягчить. Дело в том, что некоторые переменные могут оказаться неактивными в результате обработки исключения. Например, если исключение не обрабатывается в той же функции, где находится распараллеливаемый фрагмент, то ни одна переменная этой функции (при условии, что она не является синонимом для переменной более верхнего уровня либо глобальной, либо статической переменной) не будет активной, так как они не прочитаются вследствие перехода вверх по стеку. Кроме того, при возможности анализа обработчиков исключения также можно исследовать использование этих переменных в обрабатывающих исключение функциях. Если в результате генерации исключения эти переменные становятся неактивными и в вызывающем коде, то их состояние можно не отслеживать. Это позволяет оставлять их при генерации исключения в любом состоянии и не тратить вычислительные ресурсы на отслеживание изменений таких переменных. Таким образом, проведение проверки на использование в catch- и finally-блоках переменных, внешних по отношению к распараллеливаемому фрагменту, дает возможность избежать накладных расходов.

Метод решения задачи. Начнем процесс распараллеливания исходного кода программы. Распределим распараллеливаемый фрагмент на потоки выполнения. В результате разбиения получим потоки, в которых один или несколько потоков могут сгенерировать исключение.

Проблема перехвата исключения и передачи его следующему за распараллеливаемым фрагментом коду может быть решена с помощью установки одного или нескольких флагов исключения. Для выставления флага исключения код всех потоков заключается в try-блоки. Соответствующие этим try-блокам catch-блоки должны быть настроены на перехват всех исключений, как и ошибок времени выполнения (например, с использованием инструкций `catch (Throwable thr)` в языке Java [1] и `catch (...)` в языке C++ [7]). Задача обрамляющего обработчика исключений — выставление флага исключения и сохранение объекта исключения для последующей передачи в последовательный

код. После завершения всех потоков при передаче управления в последовательный код проводится проверка на установленный флаг исключения. Если флаг исключения установлен, то последовательный код генерирует исключение (или ошибку) с сохраненным объектом в качестве параметра. В этом случае исключение перехватывается тем try- или catch-блоком, который отвечает за него в исходном однопоточном коде, и логика работы программы не изменяется.

Некоторые языки, например C++11, поддерживают обмен исключениями в многопоточной среде [7]. Этот механизм также может быть применен для передачи исключения в однопоточный код.

Основная проблема — упорядочение управления при генерации исключения. Необходимо, чтобы логика обработки исключения не изменилась, несмотря на то, что вследствие преобразования кода в многопоточный может произойти переупорядочивание операций при выполнении. Для решения этой проблемы предложен следующий алгоритм.

Шаг 1. В однопоточном коде всем операторам, генерирующим исключение, присваивается временна́я метка, отражающая порядок выполнения операторов. Если один и тот же оператор участвует в формировании нескольких трасс потоков, то каждой трассе в соответствии с порядком выполнения программы для этого оператора назначается своя временна́я метка. В качестве временно́й метки можно использовать естественные признаки, например номер итерации цикла. Если один и тот же оператор может быть реализован в потоке несколько раз, то необходимо применять стратегию для присвоения временны́х меток, например, использовать номер итерации цикла или комбинировать метки с другими признаками, определяющими порядок выполнения операторов. Приведем простую стратегию для простановки временны́х меток:

1) для последовательности операторов метки проставляются последовательно;
2) для ветвления метки проставляются последовательно для каждого выполняемого блока. Так, при нумерации с k then-блоку присваиваются метки $k+1,\ k+2, \ldots, k+n$, а else-блоку — метки с $k+n+1$;
3) при наличии циклов номер итерации является множителем временной метки.

Таким образом, основным требованием к распараллеливаемому коду является его структурированность.

Шаг 2. В процессе преобразования кода в многопоточный каждому создаваемому потоку присваивается флаг генерации исключения и флаг успешного завершения. Кроме того, для каждого потока, генерирующего исключение, необходимо зарезервировать область памяти для хранения объекта этого исключения. Флаг генерации исключения

будет устанавливаться обрамляющим catch-блоком кода потока, флаг завершения — каждым потоком перед успешным завершением. Для всех потоков следует создать один общий флаг исключения, устанавливаемый вместе с флагом потока.

Шаг 3. Для каждой переменной, которая останется активной в случае генерации исключения, создать список, каждый элемент которого является парой временна́я метка–значение. Каждый поток вместо записи непосредственно в переменную должен выполнить запись в этот список.

Шаг 4. Перед каждым оператором, имеющим временну́ю метку, при генерации исключения устанавливается его временной штамп. Таким образом, после завершения всех потоков можно узнать, на какой именно временно́й метке остановился каждый поток.

Шаг 5. Если при генерации исключения поток занимал объект синхронизации, то этот объект должен быть освобожден либо в finally-блоке обрамляющего try-блока, либо другим способом, например с помощью освобождающего деструктора в модели RAII, как в языке C++.

Шаг 6. При выходе из распараллеленной секции в однопоточный код, если установлен флаг исключения, среди потоков, сгенерировавших исключение, необходимо найти поток с минимальной временно́й меткой. Значением переменной будет значение из списка, соответствующее предыдущей временно́й метке.

Предложенный алгоритм допускает выполнение операций, которые могут быть не реализованы в случае генерации исключения. Подход, при котором осуществляются операции, результат которых может быть не использован при дальнейшем выполнении программы, называется *спекулятивным выполнением*. Такой подход применяется в микроэлектронике при разработке процессоров для повышения производительности работы конвейеров. В работе [11] описаны различные оптимизации генерируемых кодов программ, связанные со спекулятивным выполнением и осуществляемые на этапе компиляции исходных кодов.

Для уменьшения размера результирующего кода, а также ускорения результирующей программы, можно вместо списка записей использовать одномерный массив, размер которого равен числу потоков. Этот прием можно применять в тех случаях, когда диапазоны временны́х меток строго разделены. Кроме того, можно добавлять в код создаваемых потоков периодические проверки на произошедшее исключение, что позволит быстро завершить работу оставшихся потоков в случае, когда один из них сгенерировал исключение.

Примеры. Рассмотрим несколько примеров распараллеливания кода цикла на языке Java с помощью предлагаемого метода.

Листинг 1. Пример цикла, использующего внешнюю по отношении к нему переменную и генерирующего исключение, приведен ниже:

```
int someVar;
// Распараллеливаемый цикл
for (int i = 0; i < n; i++) {
    throwingOperation();
    someVar = someOp();
}
```

Этот код с применением предлагаемого метода может быть распараллелен способом, показанным в листинге 2. В качестве временно́й метки используется номер итерации цикла, а вместо проверки события исключения — сравнение объекта исключения с null.

Листинг 2. Распараллеленный цикл с обработкой исключения без дополнительных оптимизаций:

```
int someVar;
// Объект синхронизации
final Object syncObj = new Object();
// Номер итерации, на которой сгенерировано исключение
volatile int iterExcNo = -1;
// Объект исключения
volatile Throwable loopThr = null;
// Список “номер итерации/записанный элемент”
ConcurrentLinkedQueue<Pair<int, int» someVarWrites = new
ConcurrentLinkedQueue<Pair<int, int»();
volatile int lastWrite;
// Запускаем потоки или применяем задачу для существующего пула
for (int j = 0; j < nThreads; j++) {
new Thread(new Runnable() {
    public void run() {
        try {
            // Разделенный на итерации цикл
            for (int i = j*nThreads; i < j*(nThreads + 1); i++) {
               throwingOperation();
               int value = someOp();
               someVarWrites.add(new Pair<int, int>(i, value);
               if (isLastWrite())
                  lastWrite = value;
            }
        }catch (Throwable thr) {
            synchronized (syncObj) {
               iterExcNo = i;
               loopThr = thr;
            }
        }
    }
}).start();
}
// Дождаться завершения потоков...
```

```
// Однопоточный код
if (loopThr == null)
    // Если никакое исключение не сгенерировано, просто запишем
    // последнее значение
someVar = lastWrite;
else {
    // иначе ищем последнюю возможную запись в списке записей
int numToFind = iterExcNo - 1;
if (numToFind > 0) {
        for (Object obj : someVarWrites) {
            Pair<int, int> pair = (Pair<int, int>)obj;
            if (pair.getFirst() == numToFind)
                someVar = iterExcNo;
        }
}
// генерация сохраненного исключения
throw loopThr;
}
```

Если использовать оптимизации, связанные с обработкой циклов, в частности массив, то результирующий код можно значительно упростить. Оптимизированный код показан в листинге 3.

Листинг 3. Распараллеленный цикл с обработкой исключения с дополнительными оптимизациями:

```
int someVar;
int threadExcNo = nThreads - 1;
final Object syncObj = new Object();
ThrowableloopThr = null;
// volatile здесь не нужен, поскольку каждый поток работает со своим
// элементом массива, а однопоточный код имеет барьер памяти
// после Thread.join() [12]
Integer someVarWrites = new Integer[nThreads];
// Запускаем потоки или применяем задачу для существующего пула
for (int j = 0; j < nThreads; j++) {
new Thread(new Runnable() {
    public void run() {
        try {
            // Разделённый на итерации цикл
            for (int i = j*nThreads; i < j*(nThreads + 1) &&
(loopThr != null && threadExcNo > j); i++) {
                throwingOperation();
                int value = someOp();
                someVarWrites[i] = new Integer(value);
            }
        } catch (Throwable thr) {
            synchronized (syncObj) {
                if (j <= threadExcNo) {
                    threadExcNo =j;
                    loopThr = thr;
                }
            }
```

```
            }
        }
    }).start();
    }
    // Дождаться завершения потоков...
    // Однопоточный код
    Int num = threadExcNo;
    // Найти последнюю валидную запись
    while (num >= 0 && someVarWrites[num] != null)
        num−;
    if (num >= 0)
        someVar = someVarWrites[num];
    if (loopThr != null)
        throw loopThr;
```

Аналогичный код может быть создан и с помощью стандартных механизмов многопоточности. Так, в языке Java для этого может быть использован интерфейс java.util.concurrent.Future, позволяющий передать исключение путем генерации исключения ExecutionException методом get( ) [6]. Аналогичный механизм присутствует и в языке C++11, где для этого может применяться класс std::future, метод get() которого генерирует исключение класса std::future_error [7]. Однако в приведенных ранее листингах потоки создавались вручную в целях повышения наглядности.

Согласно листингу 3, внутри параллельного кода не используются механизмы синхронизации, кроме случая генерации исключения. Это позволяет достичь хороших показателей параллельности полученного кода. Вследствие дополнительных проверок в условии цикла при возникновении исключения потоки, записи которых не будут учтены, сразу завершатся, что сокращает время ожидания при возникновении исключения. Также упрощен код окончательной записи в переменную. Если исключение не было сгенерировано, то накладные расходы на обработку ситуации, связанной с исключением, будут минимальны.

Тестирование. Можно рассчитать время выполнения распараллеленного цикла. Пусть цикл содержит n итераций, выполняемых k потоками, а генерация исключения происходит на итерации m. Если итерации цикла распределены между потоками так, что один поток выполняет несколько соседних итераций (поток i выполняет итерации в интервале $[in/k, (i+1)\,n/k)$, то возможны два варианта выполнения программы:

1) исключение генерируется при выполнении первого потока: $m < n/k$. В этом случае сразу завершается как первый поток, так и все остальные после получения сигнала о генерации исключения. При этом время выполнения однопоточной программы будет приблизительно равно времени выполнения многопоточной программы;

2) исключение генерируется при выполнении второго потока и потоков с большими номерами: $m \geq n/k$. В этом случае необходимо дождаться завершения работы предыдущих потоков (потоки с большими номерами можно завершить немедленно) и только после этого завершить работу потоков. Следовательно, время выполнения многопоточной версии будет равно времени выполнения одного потока.

Время выполнения однопоточной программы будет линейно возрастать при увеличении номера итерации цикла, на которой происходит генерация исключения.

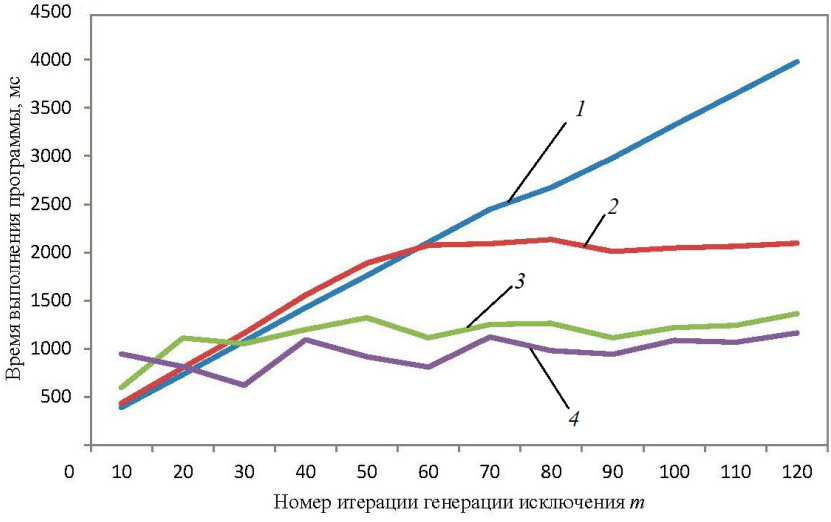
Для тестирования подхода было создано программное обеспечение на языке Java. Тестирование заключалось в прогоне цикла, содержащего генерацию исключения, с разными параметрами (в том числе, при различном числе потоков выполнения) и замером времени выполнения тела цикла как без генерации исключения, так и при его генерации на различных итерациях. Каждый прогон представлял 120 вычислений 50 000 числа Фибоначчи с использованием рекуррентной формулы, что обеспечивало полную загрузку вычислительных ядер. Тестирование осуществлено на компьютере с процессором Intel Core i7 3770K (четыре физических или восемь виртуальных ядер, 3,4 ГГц) и 16 Гб оперативной памяти под управлением операционной системы Linux с использованием Oracle JDK 1.7.45. Время выполнения программы при различном числе потоков и без генерации исключений представлено ниже:

Число потоков k	1	2	4	8
Время выполнения, мс	3976	2084	1409	1135

Зависимость времени выполнения программы от номера итерации генерации исключения m при различном числе потоков k приведена на рисунке.

Для всех рассчитанных вариантов $n = 120$. При $m < n/k$ время выполнения программы возрастает линейно, при $m \geq n/k$ время выполнения не увеличивается и остается стабильным ($m = 60$ при $k = 2$, $m = 30$ при $k = 4$ и $m = 15$ при $k = 8$). Таким образом, выдвинутые в настоящей работе предположения о времени выполнения программы подтверждены экспериментально: генерация исключения практически не влияет на степень параллелизма результирующего кода, при этом при распараллеливании можно достичь кратного ускорения программы.

Вывод. В работе исследована возможность применение метода спекулятивного выполнения для распараллеливания исходного кода, содержащего обработку исключений. Предложен новый алгоритм, позволяющий решить проблему упорядочения управления при распараллеливании фрагментов кода при генерации исключений. С использованием дополнительных приемов можно минимизировать накладные



Зависимость времени выполнения программы от номера итерации генерации исключения m при числе потоков $k = 1$ (*1*), 2 (*2*), 4 (*3*) и 8 (*4*)

расходы как с генерацией исключения, так и без генерации исключения, при этом генерация исключения не влияет на степень параллелизма результирующего кода, что подтверждают результаты тестирования.

ЛИТЕРАТУРА

1. *Bacon D.F.*, *Graham S.L.*, *Sharp O.J.* Compiler transformations for high-performance computing // ACM Computing Surveys. 1994. Vol. 26. No. 4. P. 345–420 [Электронный ресурс] URL: http://pages.cs.wisc.edu/ fischer/cs701.f00/surveys.Dec94.pdf (дата обращения: 22.03.2014).
2. *Chan B.* Run-Time Support for the Automatic Parallelization of Java Programs. University of Toronto. 2002. 110 p. [Электронный ресурс] URL: http://www.eecg.toronto.edu/ tsa/theses/bryan_chan.pdf (дата обращения: 22.03.2014).
3. *Bradel B.J.* Automatic Program Parallelization Using Traces. University of Toronto. 2010. 266 p. [Электронный ресурс] URL: https://tspace.library.utoronto.ca/bitstream/1807/26502/1/Bradel_Borys_J_201011 _PhD_thesis.pdf (дата обращения: 22.03.2014).
4. *Par4All*: From Convex Array Regions to Heterogeneous Computing / M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J.O. McMahon, Fr.-X.r Pasquier, Gr. Péan, P. Villalon // IMPACT 2012. Second International Workshop on Polyhedral Compilation Techniques. 2012.
5. *Chen M.K.*, *Olukotun K.* The Jrpm System for Dynamically Parallelizing Java Programs // Proceeding ISCA ’03 Proceedings of the 30th annual international symposium on Computer architecture. Stanford University. 2003. Vol. 31 Iss. 2. P. 434–446. DOI: 10.1145/871656.859668
6. *The Java* Language Specification. Java SE 7 Edition / J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. 28.02.2013. P. 303–311 [Электронный ресурс] URL: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf (дата обращения: 22.03.2014).

7. *Working* Draft. Standard for Programming Language C++. 2011. P. 394–401 [Электронный ресурс] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf. (дата обращения: 22.03.2014).
8. *Exception* Handling in LLVM [Электронный ресурс] URL: http://llvm.org/docs/ExceptionHandling.html (дата обращения: 22.03.2014).
9. *Skochinsky I.* Compiler Internals: Exceptions and RTTI. Монреаль, 2012 [Электронный ресурс] URL: http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf (дата обращения: 22.03.2014).
10. *Ахо А.В.*, *Лам М.С.*, *Сети Р.*, *Ульман Дж.Д.* Компиляторы: принципы, технологии и инструментарий / пер. с англ. М.: Вильямс, 2008. 1185 с.
11. *Белеванцев А.А.*, *Гайсарян С.С.*, *Иванников В.П.* Построение алгоритмов спекулятивных оптимизаций // Программирование. 2008. № 3. С. 1–22.
12. *Java* Concurrency in Practice / B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea. Addison-Wesley Professional, 2006.

REFERENCES

[1] Bacon D.F., Graham S.L., Sharp O.J. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 1994, vol. 26, no. 4, pp. 345–420. Available at: http://pages.cs.wisc.edu/ fischer/cs701.f00/surveys.Dec94.pdf (accessed 22 March 2014).
[2] Chan B. Run-time support for the automatic parallelization of java programs. University of Toronto, 2002, 110 p. Available at: http://www.eecg.toronto.edu/ tsa/theses/bryan_chan.pdf (accessed 22 March 2014).
[3] Bradel B.J. Automatic program parallelization using traces. University of Toronto, 2010, 266 p. Available at: https://tspace.library.utoronto.ca/bitstream/1807/26502/1/Bradel_Borys_J_201011_PhD_thesis.pdf (accessed 22 March 2014).
[4] Amini M., Creusillet B., Even S., Keryell R., Goubier O., Guelton S., McMahon J.O., Pasquier Fr.-X.r., Péan Gr., Villalon P. Par4All: From Convex Array Regions to Heterogeneous Computing. *Proc. 2nd Int. Workshop on Polyhedral Compilation Techniques*. IMPACT 2012. Paris, France, 2012.
[5] Chen M.K., Olukotun K. The Jrpm System for Dynamically Parallelizing Java Programs. *Proc. 30th Annual Int. Symp. on Computer Architecture (Proc. ISCA '03)*. Stanford University, 2003, vol. 31, iss. 2, pp. 434–446. Doi: 10.1145/871656.859668
[6] Gosling J., Joy B., Steele G., Bracha G., Buckley A. The Java language specification. Java SE 7 Edition, 28.02.2013, pp.303-311. Available at: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf (accessed 22 March 2014).
[7] Working Draft. Standard for Programming Language C++, 2011, pp. 394-401. Available at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf (accessed 22 March 2014).
[8] Exception Handling in LLVM. Available at: http://llvm.org/docs/ExceptionHandling.html (accessed 22 March 2014).
[9] Skochinsky I. Compiler Internals: Exceptions and RTTI. Montreal, 2012. Available at: http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf (accessed 22 March 2014).
[10] Aho A.V., Lam M. S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, & Tools, 2006. Addison Wesley, 2006, 1000 p. (Russ.ed.: Aho A.V., Lam M.S., Seti R., Ul'man Dzh. D. Kompiljatory: principy, tehnologii i instrumentarij. Moscow, Vil'jams Publ., 2008. 1185 p.).
[11] Belevancev A.A., Gajsarjan S.S., Ivannikov V.P. Construction of algorithms of speculative optimizations. *Programmirovanie* [Program. Comput. Software], 2008, no. 3, pp. 1–22 (in Russ.).

[12] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java concurrency in practice. Addison-Wesley Professional, 2006. 384 p.

Статья поступила в редакцию 25.03.2014

Романова Татьяна Николаевна — канд. физ.-мат. наук, доцент кафедры “Программное обеспечение ЭВМ и информационные технологии” МГТУ им. Н.Э. Баумана. Автор 35 научных работ в области математического моделирования, оптимизации и проектирования информационных систем.
МГТУ им. Н.Э. Баумана, Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5.

Romanova T.N. — Cand. Sci. (Phys.-Math.), assoc. professor of “Computer Software and Information Technologies” department of the Bauman Moscow State Technical University. Author of 35 publications in the field of mathematical simulation, optimization and data systems engineering.
Bauman Moscow State Technical University, 2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation.

Сидорин Алексей Васильевич — аспирант кафедры “Программное обеспечение ЭВМ и информационные технологии” МГТУ им. Н.Э. Баумана. Автор 11 научных работ в области разработки программного обеспечения.
МГТУ им. Н.Э. Баумана, Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5.

Sidorin A.V. — post-graduate of “Computer Software and Information Technologies” department of the Bauman Moscow State Technical University. Author of 11 publications in the field of mathematical simulation and software.
Bauman Moscow State Technical University, 2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation.