4. Сергиенко А.Б. Цифровая обработка сигналов. – СПб.: Питер, 2002.

5. Зверев В.А., Стромков А.А. Выделение сигналов из помех численными методами. – Н. Новгород: ИПФ РАН, 2001.

6. Головко В.А. Нейронные сети: обучение, организация и применение: учеб. пособие для вузов / под общ. ред. А.И. Галушкина. – М.: ИПРЖР, 2000.

7. http://developer.nvidia.com/object/gpucomputing.html – NVIDIA GPU Computing Developer.

А.С. Никонов[1], А.В. Русаков[2]

[1,2] Нижегородский государственный университет им. Н.И. Лобачевского

ВОСПРОИЗВОДИМОЕ ИСПОЛНЕНИЕ ПАРАЛЛЕЛЬНЫХ SCOOP-ПРОГРАММ

SCOOP (Simple concurrent object-oriented programming) [2, 3, 4] – технология для разработки параллельных программ. Она появилась как дополнение к языку Eiffel [7] и была интегрирована в EVE [8] – исследовательскую версию EiffelStudio.

Множество примеров доказали простоту использования SCOOP [3], однако разработка параллельных программ все еще сопровождается некоторыми проблемами. Одна из основных сложностей – отладка, так как поведение параллельной программы во время исполнения зависит от планировщика операционной системы. Под поведением параллельной программы понимается очередность доступа различных потоков к общей памяти. Возникает задача записи и повторения поведения параллельной программы для последующей отладки. Чтобы решить эту проблему в общем случае, необходимо модифицировать планировщик операционной системы, что невозможно, например, для коммерческих систем. В технологии SCOOP используется собственный планировщик и простая модель синхронизации, что позволяет решить задачу другим способом.

В данной работе реализована техника, помогающая пользователям распознать ошибки синхронизации в SCOOP-программах.

Данная техника может быть использована как основа для создания процедур, тестирующих параллельные SCOOP-программы.

Техника повторения параллельных SCOOP-программ. Для повторения многопоточных программ мы должны записать расписание работы потоков и повторить в точности это же расписание при следующем запуске. Расписание работы потоков – это последовательность временных интервалов. Каждый интервал содержит инструкции конкретного потока. Границы интервалов соответствуют моментам переключения контекста исполнения программы. В дальнейшем мы будем ссылаться на расписание работы потоков, полученное с помощью планировщика операционной системы, как на *физическое расписание*. Однако, используя идею *логических расписаний*, мы можем решить эту же задачу, не прибегая к помощи планировщика операционной системы.

Логическое расписание. Чтобы лучше понять идею логического расписания, рассмотрим пример простой SCOOP-программы, представленной ниже.

В примере поток *MAIN* создает поток *thread1*. Оба потока *MAIN* и *thread1* имеют доступ к общей памяти, объекту *shared_obj*. *MAIN* читает, а *thread1* читает и пишет в *shared_obj, k* и *j* – локальные переменные потоков.

```
class
                MAIN

create
                make

feature
make
local
                j : INTEGER
        do
                create shared_obj
                create thread1.make ( shared_obj )
                run ( thread1 )
                j := 20
                io.put_string ( "shared = " + shared_obj.value.out + ",
                j = " + j.out )
        end
run ( a_thread : attached separate MYTHREAD )
do
        a_thread.run
end

feature {NONE}
```

```
        thread1 : separate MYTHREAD
        shared_obj : OBJECT
end
```

```
class
        MYTHREAD
create
        make

feature
        make ( an_object : OBJECT )
        do
                obj := an_object
        end

        run
        local
                k : INTEGER
        do
                k := 5
                obj.add ( k )
        end

feature {NONE} -- Implementation
        obj : OBJECT
end
```

```
class
        OBJECT

feature
        add ( a_value : INTEGER )
        do
           value := value + a_value
        end

        value : INTEGER

end
```

На рис. 1 изображены примеры исполнения программы (физические расписания) на однопроцессорной машине.

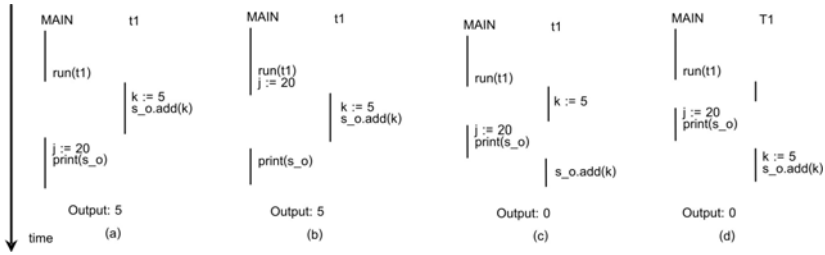


Рис. 1. Примеры исполнения многопоточной программы

Результаты выполнения программы различны и зависят от порядка доступа к общей памяти. Мы будем помещать физические расписания с одинаковым порядком доступа к общей памяти в один класс эквивалентности. В данном примере расписания (a) и (b) принадлежат к одному классу эквивалентности, (c) и (d) – к другому. Обозначим все *физические расписания* из одного класса эквивалентности как *логическое расписание*.

События синхронизации могут влиять на порядок доступа к общей памяти и, следовательно, на возможное логическое расписание. Простая SCOOP-модель не предоставляет явных примитивов синхронизации, таких как мьютексы и семафоры, синхронизация обеспечивается автоматически при каждом сепаратном вызове [2]. Поэтому мы можем рассматривать каждый сепаратный вызов как примитив синхронизации. Фактически, обязательным условием работы техники повторения параллельной программы является запись и воспроизведение последовательности событий синхронизации.

Запись поведения параллельной SCOOP-программы. Подход, применяемый для получения логического расписания, использует «глобальные часы» (целочисленный счетчик) планировщика SCOOP и «локальные часы» для каждого SCOOP-процессора. Алгоритм, работающий внутри SCOOP-планировщика, представлен ниже.

1. В начале локальные и глобальные счетчики имеют одинаковые значения (скажем, ноль).

2. Когда очередной сепаратный вызов готов к исполнению на конкретном процессоре, SCOOP-планировщик перехватывает вызов и сравнивает значение глобального счетчика с локальным счетчиком процессора-исполнителя.

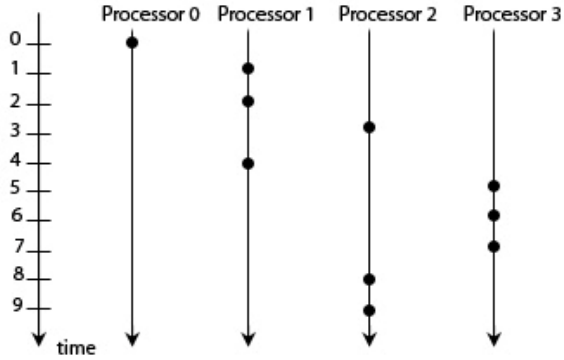
3. Если эти значения различны, планировщик только что обнаружил конец предыдущего логического интервала и начал новый логический интервал.

4. Планировщик SCOOP увеличивает значение глобального счетчика, синхронизирует с ним локальный счетчик процес-

сора и переводит в состояние «готов к исполнению» текущий сепаратный вызов.

Результат работы алгоритма – последовательность логических интервалов. На рис. 2 представлен результат работы алгоритма.

На рис. 2 временная ось ассоциирована с планировщиком SCOOP. Каждая точка обозначает сепаратный вызов, связанный с определенным процессором. Нам необходимо знать только момент, когда планировщик SCOOP обрабатывает текущий сепаратный вызов. Но не имеет значения, когда сепаратный вызов начнет физически исполняться и когда он будет закончен.



Processor 0: [0, 0]; Processor 1: [1, 2], [4, 4];
Processor 2: [3, 3], [8, 9]; Processor 3: [5, 7]

Рис. 2. Определение логического расписания

Идентификация SCOOP-процессоров. Логическое расписание не содержит достаточно информации для воспроизведения поведения параллельной программы. От запуска к запуску мы получаем различные идентификаторы потоков, а следовательно, и SCOOP-процессоров. Однако процедуру создания SCOOP-процессоров контролирует планировщик SCOOP. Поэтому мы можем обеспечить однозначную идентификацию процессора, введя ссылку на его родителя. Рассмотрим следующий пример:

```
class
        APPLICATION

create
        make

feature
        make
                do
                        create obj1.make
--creating new separate object inside a make feature
                        create obj2.make
--creating new separate object inside a make feature
                        create obj3.make
--creating new separate object inside a make feature
                end
feature {NONE}
        obj1, obj2, obj3 : separate OBJECT
end
```

Модель SCOOP гарантирует последовательное создание процессоров. Далее, конструкторы объектов могут исполняться параллельно. Но это не имеет значения, так как мы уже определили номер дочернего процессора и зафиксировали его родителя.

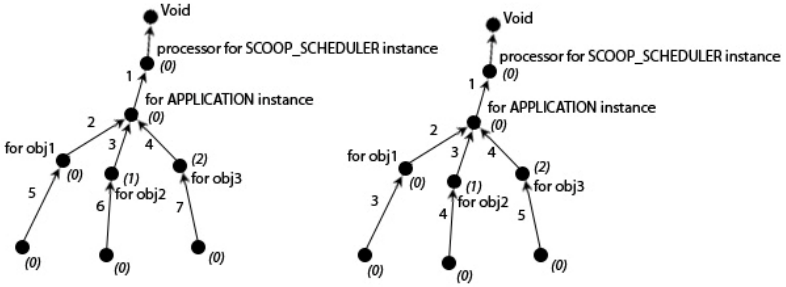


Рис. 3. Механизм создания SCOOP-процессоров. Независимо от порядка создания мы получаем одинаковую структуру процессоров

Пример создания SCOOP-процессоров в разном порядке представлен на рисунке 3. Числа над ребрами графа означают момент времени создания дочернего процессора. Числа в круглых скобках означают локальный номер процессора – для различения процессоров на одном и том же уровне дерева процессоров.

Теперь мы можем однозначно идентифицировать SCOOP-процессор при каждом запуске программы. Например, для процессора, который отвечает за объект *obj2,* мы получаем следующий идентификатор {1, 0, 0} и т.д.

Воспроизведение параллельной SCOOP-программы. После идентификации процессоров и получения логического расписания у нас есть достаточно информации для воспроизведения SCOOP-программы. Далее мы опишем алгоритм воспроизведения, встроенный в планировщик SCOOP.

1. Вначале *global_tick* должен быть установлен в ноль.

2. Увеличить *global_tick* и получить следующий сепаратный вызов из кольцевой очереди задач планировщика SCOOP. Каждый сепаратный вызов содержит SCOOP-процессор, который должен использоваться для выполнения вызова. Обозначим его как «текущий процессор».

3. Получить ранее записанное логическое расписание, *LSI,* для текущего процессора. $LSI = \{[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]\}$. Здесь используются уникальные идентификаторы процессоров для нахождения текущего процессора в логическом расписании.

Если $\exists i \in \{1, ..., n\} / [a_i, b_i] \in LSI \wedge global\_tick \in [a_i, b_i]$, тогда выполнить текущий сепаратный вызов, иначе перейти на шаг 2 и продолжить.

Теперь возможно записывать и воспроизводить поведение параллельной SCOOP-программы в целях отладки. С использованием данной техники как основного механизма становится возможным создание тестирующих процедур, которые последовательно покрывают все множество состояний параллельной SCOOP-программы.

Тестирование – наиболее важная техника для обеспечения качества программного обеспечения. Так, недетерминированное исполнение параллельной программы не позволяет покрыть удовлетворительным способом все множество состояний программы. Поэтому тестирование должно быть основано на воспроизведении конкретного программного состояния. При этом все состояния должны генерироваться определенным образом, покрывая все множество возможных состояний. В отсутствие подобной техники ошибки синхронизации трудно отследить и воспроизвести, что делает отладку и тестирование очень непростым процессом. В данной

работе был реализован подход, который может лечь в основу процедур, тестирующих параллельные SCOOP-программы.

Проект выполнен в рамках стажировки в ETH (Цюрих, Швейцария). Авторы благодарят профессора Б. Мейера (ETH), а также лабораторию Интел-ННГУ «Информационные технологии» (ITLab).

Список литературы

1. Choi J.-D. and Srinivasan H. Deterministic Replay of Java Multithreaded Applications // IBM T. J. Watson Research Center. – Hawthorne, NY, USA, 10532.

2. Meyer B. Object-Oriented Software Construction, chapter 31, 2nd edition // Prentice Hall, 1997.

3. Morandi B., Bauer S., and Meyer B. SCOOP – a contract-based concurrent object-oriented programming model / Technical report, ETH Zurich and Ludwig-Maximilians-Universitat Munchen, 2009.

4. Nienaltowski P.: Practical framework for contract-based concurrent object-oriented programming / PhD dissertation 17061, Department of Computer Science, ETH Zurich, February. 2007.

5. Leblanc T.J. and Mellor-Crummy J.M. Debugging parallel programs with instant replay // IEEE Transactions on Computers, C-36(4):471-481, April, 1987.

6. Tai K.C., Carver R.H., and Obaid E.E. Debugging concurrent Ada programs by deterministic execution // IEEE transactions on Software Engineering, 17(1):45-63, January 1991.

7. Nikonov A., Rusakov A. Научный отчет по проекту (англ.) – URL: http://se.ethz.ch/projects/andrey_nikonov/report.pdf.