The Journal of Systems and Software 85 (2012) 2519–2530

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Performance analysis of SCOOP programs

Benjamin Morandi*, Sebastian Nanz, Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Clausiusstrasse 59, 8092 Zurich, Switzerland

ARTICLE INFO

Article history:
Received 21 December 2011
Received in revised form 23 May 2012
Accepted 23 May 2012
Available online 5 June 2012

Keywords:
Performance analysis
Performance metric
Profiling
Tracing
Concurrent programming
SCOOP

ABSTRACT

To support developers in writing reliable and efficient concurrent programs, novel concurrent programming abstractions have been proposed in recent years. Programming with such abstractions requires new analysis tools because the execution semantics often differs considerably from established models. We present a performance analyzer that is based on new metrics for programs written in SCOOP, an object-oriented programming model for concurrency. We discuss how the metrics can be used to discover performance issues, and we use the tool to optimize a concurrent robotic control software.
© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Avoiding concurrency-specific errors such as data races and deadlocks is still the responsibility of developers in most languages that provide synchronization through concurrency libraries. To avoid the problems of the library approach, a number of languages have been proposed that integrate synchronization mechanisms. SCOOP (Simple Concurrent Object-Oriented Programming) (Meyer, 1997; Nienaltowski, 2007), an object-oriented programming model for concurrency, is one of them.

The main idea of SCOOP is to simplify the writing of correct concurrent programs, by allowing developers to use familiar concepts from object-oriented programming, but protecting them from common concurrency errors such as data races. Empirical evidence supports the claim that SCOOP indeed simplifies reasoning about concurrent programs as opposed to more established models (Nanz et al., 2011). The advantages of the model are due to a runtime system that automatically takes care of operations such as obtaining and releasing locks, without the need for explicit program statements.

The complex interactions between concurrent components make it difficult to analyze the behavior of concurrent programs. Effective use of a programming model therefore requires tools to help developers analyze and improve programs. Static analysis of models, e.g., Ostroff et al. (2008), Brooke et al. (2007), West et al. (2010), Nanz et al. (2008), can establish some degree of functional correctness. However, they fail to explain why a particular execution is slow, and they do not help choosing optimal execution parameters. Addressing such issues requires adapting performance analysis techniques to the context of concurrent, non-deterministic execution. Section 6 surveys existing tools that address this goal in the context of threading and various other concurrency models. They are not appropriate, however, for the semantics of SCOOP, which requires different approaches for measuring and visualization. For example, SCOOP programs go through synchronization steps to lock resources and establish conditions on these resources; a performance analyzer for SCOOP must take this into account.

We present a performance analyzer for SCOOP programs. The main contributions are performance metrics for SCOOP and a technique to compute them from event traces. The resulting tool has been integrated into the EVE development environment (ETH Zurich, 2012a), which we extended with support for SCOOP; it can be downloaded from the SCOOP website (ETH Zurich, 2012b). We evaluate the metrics and the tool on a number of example problems, as well as on a larger case study on optimizing a robotics control software written in SCOOP. To the best of our knowledge, this work is the first to suggest performance metrics for SCOOP.

This article is structured as follows. Section 2 gives an overview of the SCOOP model. Section 3 introduces the metrics and shows how to calculate them from events. Section 4 describes the tool built around the metrics. Section 5 analyzes the time overhead of the tool and shows how to optimize a concurrent robotic control software using the tool. Finally, Section 6 provides an overview of related work and Section 7 concludes with an outlook on future work.

* Corresponding author. Tel.: +41 44 632 7828; fax: +41 44 632 1435.
E-mail addresses: benjamin.morandi@inf.ethz.ch (B. Morandi), sebastian.nanz@inf.ethz.ch (S. Nanz), bertrand.meyer@inf.ethz.ch (B. Meyer).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.05.076

2. Background

This section gives an overview of SCOOP.

2.1. Introduction to SCOOP

The starting idea of SCOOP is that every object is associated for its lifetime with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing actions on objects. An object's class describes the possible actions as *features*. A processor can be a CPU, but it can also be implemented in software, for example as a process or as a thread; any mechanism that can execute instructions sequentially is suitable as a processor.

A variable *x* belonging to a processor can point to an object with the same handler (*non-separate object*), or to an object on another processor (*separate object*). In the first case, a *feature call x.f* is *non-separate*: the handler of *x* executes the feature synchronously. In this context, *x* is called the *target* of the feature call. In the second case, the feature call is *separate*: the handler of *x*, i.e., the *supplier*, executes the call asynchronously on behalf of the requester, i.e., the *client*. The possibility of asynchronous calls is the main source of concurrent execution. The asynchronous nature of separate feature calls implies a distinction between a feature call and a *feature application*: the client logs the call with the supplier (feature call) and moves on; only at some later time will the supplier actually execute the body (feature application).

The producer–consumer problem serves as a simple illustration of these ideas. A root class defines the entities *producer*, *consumer*, and *buffer*. Assume that each object is handled by its own processor. One can then simplify the discussion using a single name to refer both to the object and its handler. For example, one can use "producer" to refer both to the producer object and its handler.

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]
  – The data structure for exchanging objects between the producer and the consumer.
```

The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as *producer* will create an object on another processor; by default the instruction also creates that processor.

Both the producer and the consumer access an unbounded buffer in feature calls such as *buffer.put* (*n*)and *buffer.item*. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. Such targets are called *controlled*. For instance, in *consume*, *buffer* is a formal argument; the consumer has exclusive access to the buffer while executing *consume*.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition of the form *x.some_condition* makes the execution of the feature wait until the condition is true. For example, the precondition of *consume* delays the execution until the buffer is not empty. As the buffer is unbounded, the corresponding producer feature does not need a precondition.

```
consume (buffer: separate BUFFER [INTEGER])
  – Consume an item from the buffer.
require
  not (buffer.count = 0)
local
  consumed_item: INTEGER
do
  consumed_item:= buffer.item
end
```

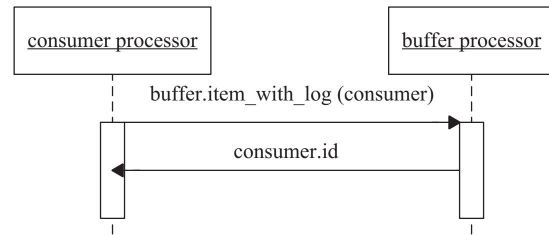

Fig. 1. A deadlock scenario based on incorrect handling of separate callbacks.

During a feature call, the consumer could pass its locks to the buffer if it has a lock that the buffer requires. This mechanism is known as *lock passing*. In such a case, the consumer would have to wait for the passed locks to return. In *buffer.item*, the buffer does not require any locks from the consumer; hence, the consumer does not have to wait due to lock passing. However, the runtime system ensures that the result of the call *buffer.item* is properly assigned to the entity *consumed_item* using a mechanism called *wait by necessity*: while the consumer usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result.

2.2. SCOOP runtime

The SCOOP concepts require execution-time support, known as the SCOOP runtime. The following description is abstract; actual implementations may differ.

Each processor maintains a *request queue* of requests resulting from feature calls on other processors. A non-separate feature call can be processed right away without going through the request queue; the processor creates a *non-separate feature request* for itself and processes it right away using its call stack. The rest of this discussion applies to separate feature calls, such as the call on the buffer performed on behalf of the consumer. When the client executes such a feature call, it enqueues a *separate feature request* to the request queue of the supplier's handler. The supplier will process the feature requests in the order of queuing.

Special attention is required in the case of *separate callbacks*, which occur for example if the buffer performs a separate feature call on the consumer, which already has a lock on the buffer. Enqueuing a feature request on the consumer could cause a deadlock if the separate callback is synchronous since the consumer may already be waiting for the buffer. Fig. 1 illustrates this issue. The solution is to add such feature requests, corresponding to separate callbacks, ahead of all others in the request queue. This ensures that consumer can process the feature request right away and the buffer can continue.

Whenever a processor is ready to let go of the obtained locks, i.e., at the end of its current feature application, it issues an unlock request to each locked processor. Each locked processor will unlock itself as soon as it processed all previous feature requests. In the example, the producer issues an unlock request to the buffer after it issued a feature request for *put*.

The runtime system includes a *scheduler*, which serves as an arbiter between processors. When a processor is ready to process a feature request in its request queue, it will only be able to proceed after the request is satisfiable. In a *synchronization step*, the processor tries to obtain the locks on the arguments' handlers in a way that the precondition holds. For this purpose, the processor sends a *locking request* to the scheduler, which stores the request in a queue and schedules satisfiable requests for application. Once the scheduler satisfies the request, the processor starts an *execution step*.

The scheduler used for this work is a dedicated thread of control (Nienaltowski, 2007). It guarantees that a satisfiable locking request

Download English Version:

https://daneshyari.com/en/article/461747

Download Persian Version:

https://daneshyari.com/article/461747

Daneshyari.com