JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

A Run-time System for SCOOP

Michael Compton, CSIRO Mathematical and Information Sciences
Richard Walker, The Australian National University

Over a period of a decade Bertrand Meyer has promoted SCOOP – a concurrency mechanism designed especially for Eiffel – but no implementation has been made widely available. In this paper we describe our initial implementation of SCOOP using the GNU Eiffel compiler. We focus on the run-time system, showing how SCOOP's synchronization mechanisms depend on the solution of a dynamic mutual exclusion problem; we give a solution using a thread and lock manager. We present a number of benchmarks, and discuss which features remain to be completed.

1 INTRODUCTION

Bertrand Meyer [16–19] has proposed a concurrency model for Eiffel, now known as *Simple Concurrent Object-Oriented Programming* (SCOOP). Although this model has been discussed and developed in detail by Meyer, no implementation has been made widely available in any Eiffel compiler. (According to Meyer, a development version of ISE Eiffel contained some SCOOP support, but this version was not made available to customers.)

The GNU Eiffel compiler, also known as SmallEiffel [7, 24], is an open-source Eiffel compiler that is also an ideal framework for experimentation in compiler research. The compiler is itself written in Eiffel and compiles Eiffel source code into C. We have used it to develop a preliminary implementation of SCOOP [8].

Section 2 explains how the SCOOP mechanism works. Section 3 provides a model that explains what a SCOOP run-time system must do. Section 4 presents details of our design and implementation. Section 5 contains a number of benchmarks of our implementation. Section 6 indicates what needs further work and discusses language issues that have arisen during the project.

2 HOW SCOOP WORKS

For convenience, we repeat here in summary form some of the definitions and rules from *Object-Oriented Software Construction*, second edition [19] (hereafter referred to as OOSC), supplementing them with our own where necessary. We begin by showing how SCOOP appears in the syntax of the Eiffel language, and then proceed to define the

Cite this article as follows: Michael Compton and Richard Walker, *A Run-time System for SCOOP*, in Journal of Object Technology, vol. 1 no. 3, Special issue: TOOLS USA 2002 proceedings, pages 119–157, http://www.jot.fm/issues/issue_2002_08/article8

notions of *processor* and *subsystem*, and show how the mechanism is supposed to work.

Syntactic Changes

SCOOP adds concurrency to Eiffel using the new keyword **separate**. This keyword may be used in class headers and in entity declarations. The following terms from OOSC (to which all page numbers refer) are used throughout the rest of this paper.

separate class any class whose class header begins:

```
separate class CLASS_NAME
```

The three qualifiers **separate**, **expanded**[1], and **deferred** which may appear before the **class** keyword are mutually exclusive and apply only to the class in which they appear, not its heirs. (page 967)

separate type any type based on a separate class. (page 967)

entity a name in the software text that denotes a run-time value. This is any attribute of a class, a variable local to a routine, or a formal argument of a routine. (pages 213, 1196)

separate entity any entity declared to be of a *separate type*, for example:

```
x: SOME_SEPARATE_CLASS
```

(where `SOME_SEPARATE_CLASS` is a separate class), or as

```
x: separate SOME_TYPE
```

(page 967)

Processors and Subsystems

Meyer introduces the notion of a *processor* to describe a unit of execution in an object-oriented system. The creation of a separate object causes the creation of a new processor to handle the processing of this new object. Concurrency in SCOOP is achieved by many processors handling the actions of different objects in parallel.

processor an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. (page 964)

This describes an abstract processor. Of course, a system is not bound by the number of its CPUs; a system may have as many abstract processors as required.

[1] In Eiffel, every type is either a *reference* type or an *expanded* type. The values of expanded types are objects; the values of reference types are references to objects. For example, the values of the expanded type `INTEGER` are integers; the values of the reference type `INTEGER_REF` are references to integers.

If sequential processing with a single processor handling the actions of all objects in one system is considered as the usual Eiffel processing model, how then is a system involving multiple processors described? We introduce a model of multiple interacting *subsystems*, so that at run time there is a system of processors, each handling the actions of one subsystem (set of objects).

Definition 2.1 A *subsystem* is a model of a processor and the set of objects it performs actions on. Within a subsystem, communication is synchronous and execution follows the usual Eiffel sequential model. Communication between subsystems is asynchronous and processing is in parallel. This potential parallelism is the result of a different processor handling each subsystem.[2]

Definition 2.2 From the point of view of an object, any object in the same subsystem is considered a *local object*.

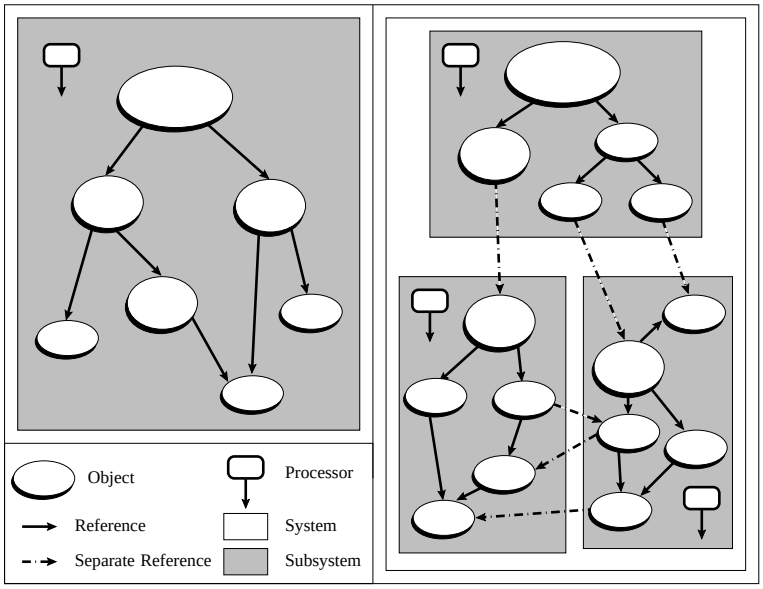


Figure 1: SCOOP system structure. The left-hand image shows the usual Eiffel system with one processor and only one thread of execution. The image on the right shows a SCOOP system with multiple processors and interacting subsystems.

Figure 1 displays the SCOOP subsystem structure. From the viewpoint of one subsystem all its objects are local, but from the viewpoint of any other subsystem all those objects are separate. This idea is explored further in Section 3. The following definitions apply to the run-time model of a program.

current object the target of the most recently started routine call. (page 1194) This definition must be extended to account for the fact that a SCOOP system may have

[2]The terminology adopted here is consistent with that used by Attali [2] to describe a similar concept.

a current object in each subsystem. The SCOOP definition states that the *current object* in a subsystem is the target of the most recently started routine call by the associated processor.

separate object any object that from the viewpoint of one object is in a different subsystem. At run time, any separate object can only be referenced (if reachable at all) through a separate entity. (page 967)

separate reference a reference to a separate object. This reference must be through a separate entity which is not `Void` and not attached to a local object. This is a restriction of the definition appearing in OOSC, as it takes into account the fact that a separate entity may, at run time, become attached to a local object. (page 967)

separate call any routine call `x.f(...)`, where the target, `x`, is a separate object. Again, this definition is refined to take into consideration the possibility of a separate entity being attached to a local object. (page 967)

An Eiffel system without SCOOP contains only one subsystem. In a SCOOP system there is initially only one subsystem. The creation of a non-separate object results in the creation of a new object in the same subsystem as the creator. However, when a creation instruction is made on a separate entity, a new subsystem and a new processor are also created. Objects may be shared between subsystems through separate references, but an object belongs to only one subsystem and is separate from all others. When passing parameters to a routine that will be executed in a different subsystem, objects which are values of expanded types are passed by copying.

Separate Calls

Consider the command `x.f(a)`, where `x` is attached to a separate object. Since `x` is attached to a different processor, execution in the current object can continue while the command `f` executes on the other processor. The current object and `x` synchronize; `x` registers the fact that `f` was called and can either start execution of `f` straight away or when the next opportunity arises. Then both the current call and `x.f(a)` can proceed concurrently.

After making the initial call, more calls may be made on `x`. After registering each call the processor making the call may continue its own execution. Multiple pending requests are served in FIFO (First In, First Out) order.

Now consider the query `y := x.some_query(...)`, where `x` is attached to a separate object. In this case, the client requires some result from the call and will be required to wait until such a result is available.

Definition 2.3 (Meyer) SCOOP-*wait-by-necessity* states that if a client has started one or more calls on a certain separate object, and it executes a call to a query on that object, that call will only proceed after all the earlier ones have been completed. Any further client operations will wait for the query call to terminate. (page 989)

(The term *wait-by-necessity* was coined by Caromel [5] and adapted by Meyer.)

Consistency Rules

Consider the following situation in a SCOOP system with two subsystems, S_1 and S_2. Object O_1 in subsystem S_1 has the following attributes:

```
x : SOME_TYPE -- not expanded or separate
y : separate ANOTHER_TYPE -- a reference
                -- to subsystem S2
```

`y` is attached to the object O_2 on subsystem S_2 (separate from the context of O_1). Assume that a routine in O_1 contains the call:

```
    ⋮
  y.some_call(x)
    ⋮
```

`ANOTHER_TYPE` has these definitions:

```
l : SOME_TYPE -- same type as x above
    ⋮
some_call(bad : SOME_TYPE) is -- traitor
 do
  l := bad -- traitor
  bad.a_call -- mistaken call
    ⋮
 end
```

Consider the consequences of the attachment of `x` from O_1 to `bad` in `some_call` of O_2 (the attachment `l := bad` is similar) and the call `bad.a_call`. The attachment results in a separate object becoming attached to an entity understood to be local and the call `bad.a_call` is understood to require local processing, when in fact, separate processing is required.

This misunderstanding, if allowed to proceed, would have the processor of S_2 executing actions on objects in S_1. This means that two threads of execution could be acting on the same subsystem, which could lead to race conditions and inconsistencies.

Meyer gives the name *traitors* to the erroneous references mentioned above. He goes on to define the following four *separateness consistency rules* to ensure that such traitors can be caught at compile time.

Note that Meyer considers that there is no harm in attaching a local object to a separate entity. We shall call these objects *false separate objects*.

Definition 2.4 A *false separate object* is a local object that at run time becomes attached to a separate entity.

Definition 2.5 *False separate processing rule*: the processing on a false separate object should continue to be synchronous, as the object is attached to the same processor, despite it being a separate entity. Allowing some other processor to handle the call would allow two streams of execution in the one object.

In order to avoid traitors, situations where traitors could be created by simple assignment must be eliminated.

Definition 2.6 (Meyer) *Separateness consistency rule 1*: if the source of an attachment (assignment or argument passing) is separate, the destination entity must be separate too. (page 973)

It must also be ensured that when passing references to separate objects, the target of the call receives them correctly as separate entities:

Definition 2.7 (Meyer) *Separateness consistency rule 2*: if an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate. (page 974)

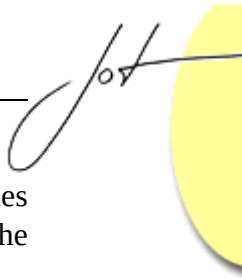
The above rules eliminate the traitors introduced in the preceding example, forcing the definitions in O_2 to be changed to:

```
l : separate SOME_TYPE
    ⋮
some_call(good : separate SOME_TYPE) is
 do
    ⋮
```

The false separate rules ensure that this feature can still be used and understood in the synchronous way for a local call, which passes a local object as an argument.

There are other sources of possible traitors, and these too must be eliminated. Any reference returned from a function call on a separate object is a source of possible traitors. It is necessary to ensure that the target of the assignment is a separate entity.

Definition 2.8 (Meyer) *Separateness consistency rule 3*: if the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate. (page 975)



Note that for the above three rules, expanded types need not be considered, as values of expanded types are passed by copy. However, in copying such values, none of the attributes (if of a reference type) should be allowed to create traitors.

Definition 2.9 (Meyer) *Separateness consistency rule 4*: if an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type. (page 975)

Reserving Objects

It can be demonstrated that, if the processors of two subsystems make interleaving calls on an object in a third subsystem, the results will depend on the order of the calls. There may be times that exclusive use of a separate object, and hence its associated subsystem, is required. In order to maintain some level of consistency, there must be some mechanism to obtain the exclusive services of an object and stop the interleaving of calls. Rather than provide a further syntactic construct, SCOOP enables this mechanism by altering the semantics of argument passing.

The meaning of:

```
some_method(x : TYPE1, ..., y : TYPE2)
```

is that the actual arguments to this call are passed to the call and used in its context. However, the feature:

```
some_method(x : separate TYPE1, ..., y : separate TYPE2)
```

indicates that exclusive locks should be obtained on x and y before the call starts.[3] Before a call can proceed, all separate objects in the actual argument list must be locked. This locking must be atomic, i.e. all locks are gained or none are gained and the processor waits until all can be obtained.

Note that false separate objects passed in an argument list need not be locked. It is useless for a processor to obtain a lock on its own subsystem.

Meyer goes further in the *separate call rule*, to state that the features of a separate object may not be called unless the object has been locked.

Definition 2.10 (Meyer) *The separate call rule*: the target of a separate call must be a formal argument of the routine in which the call appears. (page 985)

[3] It is only necessary to gain a lock on an object if its features are used in the call. A lock should be obtained on `x` only if there is some `x.feature` in the call. It is not necessary to gain locks for assignment or argument passing. This allows the passing of `Current` to separate objects without deadlock and also allows for more concurrency in some situations. Meyer [19, page 990] proposes the *business card principle*, to avoid some deadlock conditions.

The separate call rule requires that the routine call `y.some_call(x)` shown at the beginning of the section 'Consistency Rules' must now be further modified to conform to the separate call rule.

```
call_y(locked_y: separate ANOTHER_TYPE) is
 do
    ⋮
  locked_y.some_call(x)
    ⋮
 end
```

As this condition is checked at compile time, the call:

```
x.query_returning_a_separate_object(...).some_method
```

is strictly forbidden, even though in some cases it might in fact return a reference to an object that was a formal argument of the current routine.

Waiting for Other Objects

This section discusses the SCOOP mechanism for waiting for conditions on an object to become true before gaining an exclusive lock on that object.

Rather than introducing additional syntactic constructs to achieve conditional synchronization, Meyer again chooses to extend the semantics of an existing construct in the language. In the synchronous case, Eiffel's **require** clause is a correctness condition.

```
some_call(x : SOME_TYPE, ... ) is
 require
  x.some_condition
  or x.some_value = another_value
 do
    ⋮
```

The Boolean expressions listed in a require clause must be met in order for the routine to meet its contract. Should any of them evaluate to `False`, an exception is raised in the caller.[4]

Meyer extends the semantics of **require** clauses so that a clause involving a separate object becomes a wait condition. Consider the example in Figure 2, adapted from a bounded buffer program constructed during this project (assume that `ELEMENT` is an expanded type – this means that it will be passed by copy to the bounded buffer).

[4] See the *Design by Contract* concept as described by Meyer [19].

```
class SOME_CLASS
    ⋮
buffer: separate BOUNDED_BUFFER[ELEMENT]
    ⋮
  !!element.make -- create element to add
  add_to_buffer(buffer, element)
    ⋮
add_to_buffer(b: separate BOUNDED_BUFFER;
         e: ELEMENT) is
 require
  b.not_full
 do
  b.put(e)
 ensure
  b.not_empty
 end
```

Figure 2: A bounded buffer.

In this case, the require clause (stating that the buffer is not already full) is not a correctness condition which needs to be met by the client, but rather a wait condition. If the condition is not met, then the processor executing this action should wait until the condition is met, before gaining the lock on `buffer` and proceeding with the routine. The need to satisfy wait conditions and gain locks is called the *separate call semantics*.

Definition 2.11 (Meyer) *The separate call semantics*: before execution of a routine's body can start, a call must wait until every blocking object is free and every require clause involving a separate object is satisfied. In this definition, an object is *blocking* if it is attached to an actual argument and the routine uses the corresponding formal as the target of at least one call.[5] (page 996)

Eiffel's assertion-checking mechanisms may normally be turned off at compile time. This is not the case with any assertion containing separate objects. As these assertions alter the semantics of a program (rather than provide a correctness condition), they may not be turned off at compile time; they must always be used at run time.

As with the separate call rule (Definition 2.10), it is necessary to ensure that a separate object that is the target of a query in the require (or ensure) section of a routine must be a formal argument of the enclosing routine. This in turn means that calls on separate objects may not appear in the class invariant.

[5]Note that this definition has been changed from that found in OOSC as all calls, not just separate calls, must follow these semantics. Also, 'target of at least one call' is taken to mean the target of any call in the require, body, or ensure sections of the routine, not just the body.

3 A MODEL OF THE RUN-TIME SYSTEM

A SCOOP system can be modelled as a pair:

$$\Omega = \langle \mathbf{S}, \mathbf{O} \rangle \tag{1}$$

Here, **S** represents the set of subsystems:

$$\mathbf{S} = \{S_1, S_2, \ldots, S_n\} \tag{2}$$

where each S_i $(1 \leq i \leq n)$ is a subsystem. Each subsystem can be represented as a triple:

$$S_i = \langle P_i, R_i, O_i \rangle \tag{3}$$

This representation of the three essential aspects of a subsystem is the direct mapping of processors and objects as shown in Figure 1, and the new concept of a request list. The meaning of this model of a subsystem is:

- P_i models the processor which performs the execution of the subsystem;
- R_i is the model of the request queue, that is, a list of pending requests for P_i to serve (discussed further in the following subsection); and
- O_i is the set of objects in the subsystem.

The set of all objects in the system is modelled as

$$\mathbf{O} = \{o_1, o_2, \ldots, o_m\} \tag{4}$$

O_i is the set of objects in S_i, and the sets O_i, $1 \leq i \leq n$ form a partition of **O**.

Any system that implements SCOOP must be able to model these concepts correctly. It must be able to represent an Eiffel system as a set of interacting subsystems, partitioning all objects into the correct subsystem and associating a processor with each subsystem. A processor may only be active in the objects of one subsystem. However, a system may not assume that because an entity in the software text is labelled as **separate** that it will always be attached to a separate object at run time.

This model helps to show that although the software text contains the keyword **separate**, separateness is not a property of an object considered in isolation – it is relative, as shown in Figure 3. This, together with the false separate rule introduced in the subsection 'Consistency Rules', means that a system may not assume that all entities labelled as **separate** are separate objects, and instead requires some mechanism to identify the actual subsystem an object belongs to.

Calling Routines

The list of pending request calls on S_i has been modelled as R_i. What constitutes a request? Obviously routine calls (with associated arguments) should be modelled as requests and placed on the list for P_i to serve. (Attribute accesses are handled as queries.)

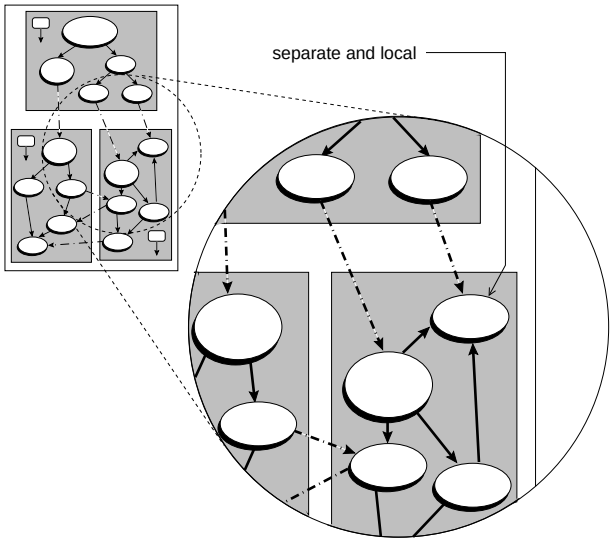


Figure 3: Certain objects, although separate with respect to some contexts, can be viewed as local objects with respect to objects in the same subsystem. For example, the labelled object is local with respect to the subsystem at the bottom right, but separate with respect to the top subsystem. If viewed in isolation, each subsystem is similar in concept to the usual Eiffel system.

Note that the SCOOP-*wait-by-necessity* mechanism (Definition 2.3) allows for many procedure calls (commands) to be issued to a separate object without waiting for their completion, but the result of any query must be waited for.

The building of the request list proceeds as follows. When some processor, say P_j, requires the execution of some feature `f` on an object in S_i, P_i and P_j must synchronize, and then `f` can be appended to the end of R_i.

The model of execution in SCOOP is similar to the classic master-slave model. A processor lies dormant until some service (modelled as a request) is called, at which time it executes the call. If, after servicing a call, another call has been appended to the request queue, this call is serviced too. The processor services all pending calls until none remain, at which point it returns to a waiting (dormant) state. A processor in a SCOOP system executes all of the pending requests in FIFO order.

A system implementing SCOOP is required to support a mechanism that permits multiple pending requests on a subsystem. This involves the (explicit or implicit) synchronization of processors to ensure mutually exclusive access to the list and the ability for the results of requests to be waited for.

Creating New Subsystems

When `!!x.creation_routine` is encountered, where `x` is a separate entity, a new subsystem and a processor for that subsystem are created. Initially, the subsystem contains only one object (the object just created). The creation of a non-separate object, from any

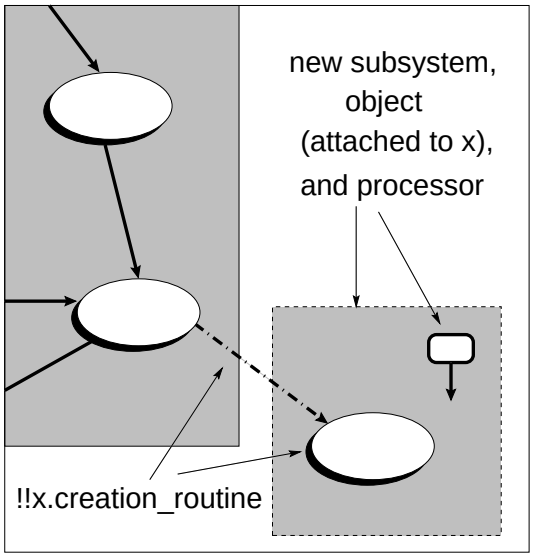


Figure 4: The creation of a separate object (in this case attached to the separate entity x) causes the creation of a new subsystem and associated processor.

object in a subsystem, say S_i, causes the creation of a local object in S_i, thus expanding the subsystem. Figure 4 shows the process of subsystem creation. At the time of subsystem creation the request queue of the new subsystem should contain only one pending request, namely, the creation routine (constructor) of the new object.

A SCOOP system is required to have the ability to create new subsystems and processors during program execution; these subsystems may grow (in the number of objects) as execution proceeds.

Once Routines

An aspect of concurrency not dealt with in OOSC is the semantics of Eiffel's **once** routines in a concurrent context. There are two basic options that should be considered.

1. Once routines are executed only once for the entire system. Each once function is executed once by the first processor to call the function and after this all other processors use the return value from this first execution.
2. Once routines are executed once for each subsystem. The first time a processor encounters a once routine, the routine is executed. Any result from a once function is used in that subsystem alone, irrespective of the results of once functions in other subsystems.

Our model can be used to deduce which option is correct in the context of SCOOP. The actions of once functions are considered, as they can create a problem in SCOOP systems.

The choice will be made by considering a simple example. Assume that a class in an Eiffel system contains the following once function:

```
some_function(...) : RESULT_TYPE is
 once
    ⋮
```

Assume that RESULT_TYPE is a non-separate reference type. Consider what would happen if Option 1 were accepted as the correct semantics. Processor P_i in subsystem S_i reaches the call to execute some_function first and performs the call, producing some local object, say o_i, of type RESULT_TYPE. If processor P_j in subsystem S_j then attempts to call the function, the result has already been computed and the object o_i is returned. However, object o_i is local to S_i and thus the reference to o_i in S_j is a traitor. This option is obviously not correct.

Option 2 is correct in this context. Any result from a once function calculated in S_i is local to S_i and has no impact on the results of once functions calculated in S_j. Our investigation shows that these semantics are correct; whether or not this is the intended semantics is an open issue and it is not the intention of this work to assert Eiffel's evolution in this context.[6] Note that if the once function returned a separate or expanded result, the situation explained above could not occur. A non-separate reference result is the only problematic case and resolving this case is enough for correctness in all situations.

There is a further issue raised by once functions in the context of SCOOP. Definition 2.9 stated that values of expanded types can only be passed between subsystems if they contain no non-separate reference entities. A refinement of this rule is required. If an expanded class contained the once function given above, and an instance of this class were passed between subsystems (as an argument to or result from a separate call), a traitor would have been created through the once function.

Definition 3.1 The new *Separateness Consistency Rule 4*: if an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute **or once function** of a reference type.

This is not the end of the matter. The class GENERAL, from which all classes inherit, contains a number of once functions that provide opportunities to create traitors and inconsistencies. For instance, one of these problems is io (an instance of the library class STD_INPUT_OUTPUT). A special semantics could be added for io as the one I/O stream is shared by all classes, i.e. all access to io could be implicitly synchronized by the compiler. We developed a special class SYNCHRONIZED_STD_INPUT_OUTPUT and an attribute sio to allow for the correct use of the I/O file stream. This may be a

[6] Meyer has proposed extensions to the Eiffel language to allow for further options for once functions: per-thread, per-system, per-object, per-time-period, and others. Note the per-system extension applies to a per-installation of Eiffel, and per-thread applies to ISE's threading mechanism, not SCOOP. This extension does not cover the case presented above and thus an understanding of this issue is important for a correct definition of once functions in the context of SCOOP.

better option, as it makes the programmer explicitly aware of the issue. This option also removes some race conditions that are possible when using the normal `io`, even when the user could believe that they were doing so correctly. A further option could be to make `io` a separate object, but this would necessitate obeying the separate call rule for every call on `io`.

There are other inconsistencies in `GENERAL` and `ANY` that break a variety of the SCOOP rules. A work-around was constructed for each of these rules to allow this project to continue, but each constitutes an issue that needs to be resolved in further development of SCOOP.

Resource Allocation

A request for a lock on an object is in fact a request for a lock on the subsystem in which the an object resides and a request for the exclusive right to issue calls to the associated processor. In SCOOP, locks will be granted in a first come, first served order.

A simple example will give further insight into the locking aspect of the mechanism.

```
a_call(x : separate SOME_TYPE) is
 do
    ⋮
  another_call(x)
    ⋮
 end
```

This call makes a request to lock `x` and then another request while `x` is still locked. There seems no reason why such a lock should be refused. Mechanisms such as the Java concurrency mechanism allow this style of recursive locking. Personal communication with Bertrand Meyer confirmed that this is the correct semantics to use for SCOOP as well. The recursive locking rule is introduced for this reason.

Definition 3.2 The *recursive locking rule*: if some processor has a lock on an object and attempts to gain a further lock, it will gain this recursive lock. Each unlock rolls back one level of the recursive locking. The object is not actually unlocked and available for use by other processors until unlocking has reverted back to the highest level.

Mutual Exclusion

We now introduce a model of the SCOOP locking problem which serves to define the problem and identifies a set of properties that must be satisfied.

Figure 5 shows that the SCOOP locking problem can be described in a manner similar to the descriptions of mutual exclusion problems found in the literature [4]. As we will

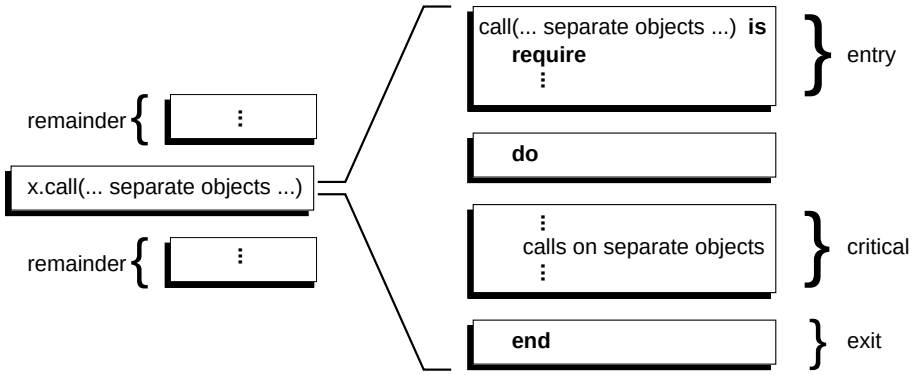


Figure 5: It is possible to model the locking problem given by Meyer's SCOOP as a mutual exclusion problem. Here Eiffel code (a routine) is broken down into the usual remainder, entry, critical and exit sections.

later be using an algorithm by Rhee [21] we use his terminology and adapt it to represent the SCOOP problem. A processor's execution cycles through four stages: remainder, entry, critical, exit, and then returns to remainder.

remainder The remainder section is the non-critical section. Its code does not make any calls on separate objects.

entry The entry section contains the code provided by the mutual exclusion solution to provide exclusive access to separate entities. The entry section in the SCOOP problem waits, if necessary, for any separate preconditions (wait conditions) to be satisfied and gains locks on all separate actual arguments to the call.

critical The critical section contains the code that requires the protection of mutual exclusion. This section contains calls on separate objects.

exit The exit section contains the solution-provided code to release the resources exclusively obtained in the entry section, including one level of recursively-held locks.

During execution of a SCOOP program, there is a subset of the processors that requires exclusive access to a subset of the resources (subsystems). These resources need to be acquired before executing a critical section.

When a processor leaves its remainder section, it selects some non-empty set of resources and some possibly empty set of conditions. Let $d_i^{t_1}$ be the resource requirements of processor P_i leaving its remainder section at time t_1. In order to leave the entry section (and thus enter the critical section), P_i must acquire all resources and have all conditions satisfied. This must be done atomically (from the viewpoint of other processors in the system). The processor must acquire all resources and satisfy all wait conditions in a single operation. If not, it must acquire none and wait until all can be gained.

This problem contains two interesting additions to the usual mutual exclusion problems.

1. The problem considered here is nested. While in its critical section, P_i may select another possibly intersecting set of resources $d_i^{t_2}$. For this set of resources P_i enters another entry → critical → exit cycle. The total resource requirements of a processor P_i at some time t is modelled as $d_i^t = d_i^{t_1} \cup d_i^{t_2} \cup \ldots \cup d_i^{t_n}$ for all the nested resource sets $1 \ldots n$ it has acquired.

2. P_i may select some set of conditions $C = \{c_1, c_2, \ldots, c_m\}$ which must be met before it leaves the entry section.

A common way to describe the conflicts between processors in such problems is to use a conflict graph [6, 21]. In such a graph, the nodes represent processors and an edge in the graph represents a conflict between processors. Nodes that share an edge have conflicting resource requirements and may not enter their critical sections at the same time. The conflict graph in this problem is dynamic. There is no prior knowledge of the number of processors, their resource requirements, or the conflicts between processors; two processors P_i and P_j conflict at time t if $d_i^t \cap d_j^t \neq \{\}$.

In solving such problems it is necessary to ensure some liveness and safety properties. In this case, we require the usual mutual exclusion properties of absence of individual starvation, and freedom from deadlock.

Additionally, we require that a *concurrency* condition be satisfied. Informally, if P_i is in its entry section and there are no conflicting processors, then P_i should enter its critical section in some finite number of its own steps. Such a condition stops the underspecification that would allow an overly restrictive mutual exclusion algorithm to solve the problem by only allowing one processor to be active at any one time.

Note the ubiquity of synchronization in the system. There are no explicit synchronization constructs, which means that any object can contribute to synchronization. It has already been shown that any object (except for values of expanded types) can be shared between subsystems and can thus be used as a separate object from some context.

The problem considered here is a type of dynamic resource allocation problem, where the set of processors, their resource requirements and the conflicts between processors are unknown [21]. The inclusion of conditions to the problem gives it aspects similar to the general mutual exclusion problem GRASP [13]. The interaction can also be described as a nested transaction system.

In general, it would be difficult to show that a run-time system always solves the resource allocation problem presented above without deadlock. The programmer may introduce deadlock into the program; compile-time deadlock detection is possible in some cases but this option is not considered here. The run-time system must not deadlock, however, when the user program is deadlock-free; the run-time system must satisfy the above resource allocation constraints.

4 DESIGN AND IMPLEMENTATION

SmallEiffel already has a stable run-time system. The performance of some aspects of this system has been evaluated [24]. The SCOOP model affects only the interaction between subsystems; within a subsystem processing is exactly as it would be in a normal Eiffel system. Because of this, as much of the existing run-time system as possible should be maintained when integrating SCOOP.

SmallEiffel requires a platform with an ANSI C compiler and a POSIX run-time system.[7] We have tried to add SCOOP to the compiler in such a way as to maintain the broadest level of platform compatibility as SmallEiffel. The only new library calls are calls to routines defined in the POSIX standard [10], or calls to routines introduced into the run-time system. The target machine could be viewed as a POSIX *virtual machine*. This necessitates the use of POSIX threads to provide the concurrency aspects of the system. In some ways this is similar to work by Foster and Taylor [9], who defined the notion of an abstract concurrent machine. The interleaving of thread executions provided by the pre-emptive scheduling of POSIX threads is assumed. This means that the system has little to no control over the scheduling of threads. For this work we have chosen to restrict ourselves to uniprocessor and symmetric multiprocessing architectures.

Design Goals

Our goals in the development of this system were:

low run-time costs There should be no extra run-time cost for non-SCOOP programs; in a SCOOP system the processing described for each subsystem should remain exactly as for the normal Eiffel case. A small additional cost for SCOOP features such as communication and locking is expected. Experiments in Section 5 show how some of these costs were reduced.

small amount of introduced code The additions to the run-time system should be kept as small as possible.

low memory cost Java systems are often concerned with the memory cost introduced by concurrency (every object requires a monitor in Java). The aim for SCOOP should similarly be to introduce only a small memory overhead to the Eiffel run-time system.

generality The system must continue to support all of Eiffel's features, including dynamic binding, polymorphism, inheritance, genericity, once routines, and all the SCOOP requirements given in detail in the previous sections. The system should also be general enough to work in each of SmallEiffel's optimization modes and be complete in all respects.

[7] Although SmallEiffel can also generate Java byte code, only the production of C code was considered for this project.

On the issue of *generality*, we have not considered inter-subsystem exceptions and garbage collection in this project. See the subsection 'Future Work' for further comments on this issue.

System Structure

Because of the concurrency constraints introduced by SCOOP, there must be the ability to run the processors of multiple subsystems in parallel, or at least create the illusion of this. To satisfy this constraint a POSIX thread (processor) is associated with each subsystem and an execution environment that this processor acts in is defined. The execution environment described here is similar to a model derived for a Java system by Agesen [1]. Figure 6 shows the structures for processors and subsystems.

```
processor {
 pthread_t thread_id
 subsystem subsyst
 pthread_mutex_t thread_mutex
 pthread_cond_t thread_cond
 long time_stamp
 request_ptr requests_pending
 request_ptr last
}
```

```
subsystem {
 ident domain_identifier
 processor executer
 long lock_count
 processor current_locker
 waiter_ptr wait_queue
}
```

Figure 6: The left hand structure represents the execution environment of a processor. The right hand structure shows the modelling of a subsystem in the SCOOP run time.

Synchronization in the system is achieved through locking (argument passing in the source program) and through results returned from separate calls. The term *synchronized on* is used to indicate this co-ordination between processors.

Our design satisfies the requirement that it be a system composed of subsystems, containing a processor for each subsystem. Section 3 showed that it was necessary for a SCOOP system to be able to discover the subsystem any object was in. How then should O_i (the set of objects in subsystem S_i) be represented? One solution is to associate a set of objects with each subsystem, but this option is bulky and unscalable. A better solution is to create a mapping $M : o \rightarrow S_i$ that maps any object to the subsystem it resides in. The most obvious way to achieve this is to add a pointer in each object. The pointer indicates the subsystem that the object resides in. This allows a processor to decide at run time if a routine call requires local sequential processing or the concurrent processing of a different processor. Note that this test need only be performed for calls on separate entities, as non-separate entities will always require local processing. (This can be determined at compile time.) Remember that at run time a local object may become attached to a separate entity. Therefore, a test is required by the false separate rule before invoking a routine on a separate entity.

An issue in the construction of concurrent Java run-time systems is the extra memory needed for concurrency. This is because any object may be locked independently and so may require an associated mutex [14]. Many techniques have been developed to reduce this cost, including lazy mutex creation, mutex compression, and lock reduction [1, 3, 14]. The cost in our SCOOP system is slightly less (4 bytes for a pointer to a subsystem). However, in large systems this is still a significant memory overhead.

Perhaps this cost could be reduced. First, consider expanded types. Values of expanded types are passed by copy between subsystems and therefore can not be referenced from another subsystem (values of expanded types are not reachable through separate references). Thus a processor knows that all values of expanded types it ever encounters are in its subsystem and require local processing. This in turn means that values of expanded types do not require a mapping to the subsystem they reside in. Second, recognize that only a subset of all types is ever synchronized on. Specifically, this set consists of all separate types and all reference types passed between subsystems, through argument passing, or as the result of a separate query. Perhaps SmallEiffel's type inference algorithm [7] could be used to discover all these types. The mapping only needs to be added to all objects of these types. Currently, only a basic approach to this idea is implemented and a more sophisticated approach could remove more of this memory cost.

Calling Routines and Accessing Attributes

The SCOOP model includes the requirement that any processor be able to handle multiple pending requests from other processors (see the subsection 'Calling Routines' above. To satisfy this requirement, a queue of requests yet to be served (shown as the attribute `requests_pending` in the processor structure of Figure 6) is associated with each processor. This models requests as a queue of functions and marshalled arguments.

In order to make a call on another processor, the calling processor must marshal the arguments (pack them into a predefined data structure for the routine), add a reference to the routine and then attach this package to the end of the `requests_pending` queue of the appropriate processor. The use of a pointer (shown as the `last` attribute in the processor structure of Figure 6) to the end of the `requests_pending` queue makes this attachment an operation requiring constant time. The actions of both processors (caller and callee) are shown in Figure 7.

The concept of marshalling arguments and sending them off to another processor to handle the routine is used in many systems, including Opus [15] and Eiffel// [5]. The method of attaching a packaged routine/data element to a request list in this way is similar to the approach taken in Eiffel//, except that the mechanism described here does not require the co-operation of both processors. In fact, the callee may continue to process other calls while new requests are attached to its list of pending requests.

Figure 7 shows that after attaching a request to the list, a processor may continue its own operations. This mechanism allows for concurrent execution by multiple processors. An experiment presented in the subsection 'Macro-benchmark' shows that subsystems

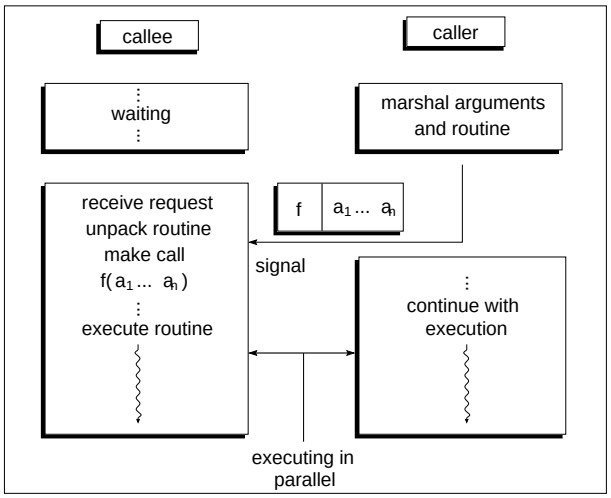


Figure 7: Procedure calls in different subsystems may proceed concurrently because different processors handle each call. This figure shows the caller and callee of a routine executing in parallel, after the appropriate routine and arguments have been passed to the callee.

can execute efficiently in parallel.

Returning Values

Returning values from a function is achieved through an extension of the general routine calling mechanism described above. The SCOOP-*wait-by-necessity* mechanism (Definition 2.3) shows that the client must wait for the result of a separate function.

Our system uses a pointer to a known result location to allow for return values from separate functions. The client of the function places the address of a memory location for the result in the marshalled arguments structure. After making the separate call, the client then waits for the completion of the function. After computing a result, the processor serving the function places the result in the location indicated by the marshalled list; the client is then signaled to indicate that the result is ready. At this time the processor that served the function will return to a waiting state and the client continues to process. (There can be no more requests to serve because of the separate call rule and because no requests are issued after a function is called.)

Both simple assignment and passing function results as arguments are supported in the system:

```
y := x.separate_function_call(...)
z.a_call(..., x.separate_function_call(...), ...)
```

This mechanism also integrates cleanly with SmallEiffel's mechanism for handling dynamic dispatch [24].

Subsystem Creation

When `!!x.creation_routine` is executed (where `x` is a separate entity), both a new subsystem and a new processor must be created. This is achieved by allocating the memory to represent `x` and attaching to `x`'s subsystem pointer a newly created subsystem. The new subsystem is created by allocating the necessary data structures for the subsystem and processor. A processor (POSIX thread) is started for the new subsystem; this processor proceeds to execute the creation routine. Letting the new processor handle the creation routine means that the execution of the creation routine may proceed concurrently with the execution of the processor that called `!!x.creation_routine`. As the new object is already attached to `x`, the creator of the new subsystem may begin to issue calls on the newly created subsystem (through `x`) even before the creation routine has completed, after gaining necessary locks.

Once Routines

The SmallEiffel compiler achieves a once semantics for single threaded programs by associating a flag with each once routine. The flag is set on the first execution of the routine. Once functions also have a result associated with them; this result is set on the first execution of the function. If the flag is set, the caller of a once function will receive the result previously computed.

This concept is extended by associating a key and thread-specific data with each once function.[8] The data associated with the key is a flag for a procedure, and a flag and result for a function. A thread uses a once routine's key to look up the flag and data before executing the routine. If the flag is not set, it performs the only execution of the routine and sets the flag. If a result is to be returned it is placed in the key-accessed data package. The body of the routine is not executed on future calls to the routine.

Silva [22] developed a SmallEiffel concurrency system which uses a combination of thread-specific data and linked lists to implement once functions. A thread would look up its identifier from a key and then search through a linked list of (identifier/data/value) elements for the once routine until the element found matched the identifier. This option is not scalable as the number of threads increases, nor is it as fast as the method described here.[9]

An issue to consider with the given implementation of once routines is the limited number of thread-specific data keys that are available according to the POSIX standard. Currently, the lower bound on this value is 256 keys; this would limit a program to 255 once functions (one key is used for the processors in the system to store their execution

[8] Thread-specific data is a POSIX threads mechanism that allows a thread to store a data element accessible by a key. Threads may store different data elements for each key. Any thread asking for the data associated with a particular key, will be returned the data it associated with the key, regardless of the data stored by other threads.

[9] The run-time cost of this implementation is at least as great as that of the the key lookup, plus locking and traversing of a linked list.

environment). This is a considerable limit; for example, the compiler itself contains 220 once functions. An alternative is to create a hash function mapping a thread's identifier to some storage location for each key. Note that both the Linux and Solaris implementations of POSIX threads allow for 1024 keys.

Resource Allocation

The important issue of resource allocation was presented in the subsection 'Resource Allocation' in the previous section. Here we discuss those aspects of the SCOOP run-time system that satisfy the various properties defined for the SCOOP resource allocation problem. Our solution to the problem is based on the distributed queue approach of Rhee [21]. Rhee's algorithm is not in itself sufficient to solve the problem at hand, so we present a modification that does, based on a centralized 'lock manager' instead of the original distributed managers.

A Lock Manager

Rhee's algorithm to solve the dynamic resource allocation problem is based on the notion of a distributed queue. Processors entering the entry section are placed on the queue of each resource they require. When a processor reaches the front of all the queues it is in, it has obtained exclusive access to all required resources. Upon entering the exit section, a processor is removed from all queues, and processors behind it move forward in the queue. Rhee proves that this approach satisfies liveness and safety properties similar to those presented in the subsection 'Mutual Exclusion'.

The algorithm considered here differs slightly. The SCOOP solution is based on a centralized 'lock manager' that allows the queues to be manipulated in a different way. A *wait queue* is associated with each subsystem. (The subsection 'Resource Allocation' in Section 3 showed that locking means gaining exclusive access to subsystems.) A processor attaches itself to a wait queue only if it can not acquire all required resources, otherwise it acquires resources and continues to the critical section. A processor on wait queues gains exclusive access to all resources once it is able to gain a lock on each resource that it requires. We now present a more detailed description of the algorithm.

Upon encountering an entry section, a processor builds the set of resources it needs to lock for this call and enters the lock manager. The lock manager is a module containing routines used for acquiring and releasing locks. Only one processor is ever active in the lock manager at any time; access to the lock manager is protected by a mutex.

There are two paths to resource acquisition, which we now describe in terms of processor P_i entering an entry section with a set of resource requirements. P_i begins by gaining exclusive access to the lock manager and obtaining a timestamp. P_i then attempts to gain each lock in turn.

- If all resources are free (unlocked) then P_i locks all required resources. P_i then

unlocks and leaves the lock manager, and begins to execute the critical section.

- If any of the resources is already held by another processor, P_i places itself on the wait queue for one of these subsystems. (Wait queues are ordered by timestamp.) It unlocks all resources acquired in this locking sequence, unlocks the lock manager and waits on its condition variable for a signal. When awoken, it repeats its attempt to lock all the required resources. (Figure 8 shows the lock queues.) This algorithm differs from Rhee's algorithm, where a processor is placed on all wait queues; this modification is possible because of the shared memory abstraction.

If an object attempts to lock a subsystem on which it already has a lock, it gains a recursive lock on the subsystem. Each processor records the depth of recursive locks it holds on each subsystem.

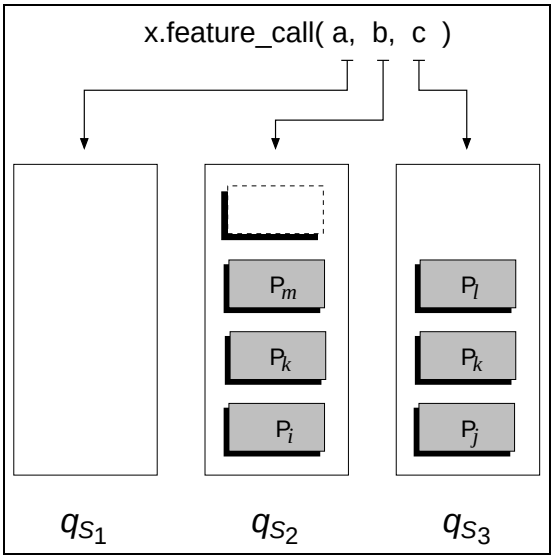


Figure 8: Three wait queues (q_{S_1}, q_{S_2}, q_{S_3}) for subsystems S_1, S_2 and S_3 respectively. Objects a, b and c are in subsystems S_1, S_2 and S_3 respectively. The processor executing the call was able to gain a lock on S_1 but not S_2 so it removes the lock on S_1 and places itself on the wait queue for S_2.

When unlocking resources, P_i performs the following actions:

- For each resource to be unlocked, P_i unlocks the resource and, if the object is to be released (zero locking depth), builds an ordered list of all processors that have been waiting on these objects. P_i signals the condition variable associated with the first processor and then waits on its own condition variable. The processor thus woken again attempts to acquire all resources (its actions are summarized by the locking protocol above); whichever path it follows (gaining locks or waiting again), it signals the next processor on the ordered list. This process continues until the last processor on the list is reached. This processor signals P_i, which wakes up, unlocks the lock manager, and returns to its normal execution. After the unlocking sequence other processors may enter the lock manager.

Abstracting locking in terms of fast and slow methods of lock acquisition, where the slow method requires communication between threads on wait queues, is similar to the approach described by Agesen [1]. The list built by the unlocking processor is called the HANDOFF-list and the process of signalling each processor in turn is called the HANDOFF-protocol. Each processor woken during the HANDOFF-protocol has exclusive access to the lock manager despite not currently owning the lock manager mutex. Thus there is always only one thread of execution in the lock manager. This means that actions on the wait queues are implicitly serialized.

The order of processors on the wait queues never explicitly changes (processors don't get a new time stamp while on the wait queues). However, a later processor may overtake an earlier one if it can gain all its resources while the earlier processor is waiting on some other resource. Some resource allocation algorithms do not allow such overtaking, but it is required in this case as the earlier process may be waiting on a lock held by the later one (because of the nested locking), so deadlock would occur if overtaking were not allowed.

The pseudo code algorithms for the entry and exit sections are given in Figures 9 and 10. Multiple instructions written on one line, separated by a dash (—), indicate a group of instructions executed atomically.

```
lock_resources(S)
/* S is the set of resources to lock */
  lock(LockManager)
  get timestamp
  i := 0
  while i < |S| do
    i := i + 1
    trylock(s_i)
    if could not lock s_i
      add self to s_i wait queue
      for j:= 1 to i do unlock s_j end
      unlock(LockManager) — signal HANDOFF — wait
      i := 0
    end
  end
  signal HANDOFF
  if this subsystem still holds a lock on LockManager
    unlock(LockManager)
  end
```

Figure 9: The entry section of the resource allocation algorithm for the SCOOP run-time system.

```
unlock_resources(S)
/* S is the set of resources to unlock */
  lock(LockManager)
  foreach s_i ∈ S
    unlock(s_i)
    if no recursive lock is held on s_i
      add s_i wait list to HANDOFF-list
    end
  end
  signal HANDOFF — wait
  unlock(LockManager)
```

Figure 10: The exit section of the resource allocation algorithm for the SCOOP run-time system.

Wait Conditions

The entry protocol was not fully described above. Part of the entry protocol involves a processor (possibly) waiting on some conditions on the separate objects before locking them. This section describes the aspect of the entry protocol that deals with wait conditions.

Before checking the wait conditions of some separate object, a processor must first lock the subsystem associated with that object. There are several reasons for this. First, it is required before the function can be called. Second, if a lock were not obtained, this would introduce a race condition – checking the conditions and then obtaining a lock would allow some time period where another processor could alter the value of one of the conditions. Third, any one wait condition may contain multiple separate objects and each of these must be locked.

The process for checking wait conditions is an extension of the locking mechanism described above. This extension must fit in easily with the normal locking mechanism, as it may be required in some situations and not others.

- A processor locks all resources (as normal) and then checks each wait condition in turn. If any of the wait conditions are not satisfied, it releases the locks obtained and places itself on wait queues as normal. Other processors that later obtain the resources may now have the ability to change the condition and signal the waiting processor.

When waiting on a wait condition for a recursive lock, the processor must release all recursive locks and regain them when the conditions are satisfied. If it did not release recursive locks, no other processor could obtain the lock and change the conditions, leading to deadlock.

This method allows a processor waiting on a condition to wait on its condition variable (in a wait queue) until some other processor takes the lock, changes the condition, and then unlocks the resource.

When waiting for conditions, a processor may have to be placed on multiple queues. Suppose that the wait condition was

```
x.some_value = y.some_value
```

where `x` and `y` are separate objects on different subsystems. The processor wanting to satisfy this condition would need to wait on both `x` and `y`, otherwise it could miss a change in one that satisfied the condition.

The algorithms for wait conditions are shown in Figures 11 and 12.

```
wait_resources(S)
/* S is the set of resources to wait on */
  lock(LockManager)
  foreach s_i ∈ S
    release all recursive locks for s_i
    add s_i wait list to HANDOFF-list
  end
  signal HANDOFF — wait
  unlock(LockManager) — wait
  lock_resources(S) /* regain locks */
  foreach s_i ∈ S
    regain all recursive locks for s_i
  end
```

Figure 11: The section of code in the lock manager that handles waiting on subsystems.

Exiting the System

A system's exit condition was not defined in OOSC. We have decided that the system will exit when all processors enter a dormant state. Of course, if all processors are dormant, then there is no way to detect this dormant state. Therefore before entering a dormant state each processor checks to see if all others are waiting, and if so, exits the system.

The model used in this project allows the initial thread of the system to enter a waiting state and lets other processors call actions on it after it becomes dormant. In this model the start system may have requests called on it, after passing a reference to one of its objects to other subsystems. This model conforms with general Eiffel principles.

Summary

This section presented the architecture of the SCOOP system we have constructed. This description does not constitute a design of all elements of the system, only a description of

```
check all correctness conditions
if any fail, raise an exception in the caller

lock_resources(separate objects)
repeat
  foreach wait condition
    check the condition
    if the condition is not met
      wait_resources(separate objects)
      break /* back to the outer loop */
    end
  end
until all wait conditions satisfied

/* continue with routine */
```

Figure 12: The code compiled into each **require** section (precondition) of a routine if it contains wait conditions. Note that the normal ensure code for correctness conditions is maintained.

the major architectural features. Some of the simplicity and the elegance of the approach may not be apparent in this description. Note that the amount of code introduced into the system was minimal. No extra code is introduced for objects that are never shared between subsystems, nor is any code introduced for routines and features not used between subsystems. The amount of code introduced for each SCOOP call is only 9 (C code) lines plus one instruction for each argument to the call. Separate once routines introduce less code. The locking and thread management code (included as a C header file) constitutes the bulk of the new code. Extra memory requirements are limited to the bookkeeping for each processor, and one pointer for some objects.

5 BENCHMARKS

This section is concerned with the process of carefully benchmarking and evaluating the modified run-time system. We present micro-benchmarks showing the performance of individual aspects of the system, and the results of experiments used to investigate and enhance the system. A macro-benchmark of a ‘real’ program is used to assess overall efficiency. Compile-time cost is not greatly affected by our modifications and therefore compile times are not shown. The execution time of an action is referred to as the *cost* of that action.

The significant differences in the implementations of Java and other Eiffel run-time systems (the details of which are often unpublished) means that providing meaningful comparisons between different run-time systems (in different languages) is a difficult task. The SPECJVM benchmarks [23] were introduced to provide a standard set of benchmarks

through which the performance of Java run-time systems may be evaluated. There exists no such option for Eiffel. Also, the marked difference in the workings of SCOOP and many other object-oriented concurrency mechanisms would make comparisons difficult. For these reasons, this project aimed to confine itself to benchmark accurately the performance of the SCOOP system, and not to compare results between systems. These arguments and benchmarking strategies are in line with those presented in the literature [20].

Three benchmark machines were used. The inclusion of two symmetric multiprocessing (SMP) machines has allowed the evaluation of the mechanism in the presence of real parallelism, as well as the simulated parallelism of single CPU machines. The benchmark machines are listed in Table 1. These machines are all representative of the platforms targeted by the SmallEiffel compiler and thus also by the SCOOP compiler.

label	CPUs	CPU and speed (MHz)	memory (MB)	OS
A	1	600 (AMD-k7)	256	Linux
B	2	350 (Pentium II)	128	Linux
C	4	256 (UltraSPARC-II)	2048	Solaris

Table 1: The three computers used in the benchmarks of this section. Each machine will be referred to by the label given here.

Note that SCOOP is not aimed at high-performance parallel machines or for use in fine-grained parallel programming. It is, however, important to investigate the performance aspects of the system to ensure that the cost of concurrency does not outweigh its benefits, and to identify areas for improvement.

Micro-benchmarks

Our discussion of benchmarking begins with a number of micro-benchmarks of individual aspects of the run-time system. The aim in these benchmarks is to investigate the relationships and timings of individual elements. As a comparison between machines is not important for this discussion, these benchmarks show results only for machine *A*. Though the individual results differ on each machine, they show comparable trends.

This investigation can be used to identify areas of future work and bottlenecks in the system. An investigation highlighting aspects of the system is consistent with benchmarks performed for other concurrency mechanisms [1, 12, 20]. Areas of interest in SCOOP are: the locking algorithm, the creation of new subsystems and processors, the signalling time between threads, and the cost (for both caller and callee) of calling concurrent routines.

Subsystem creation and requests

Timings were obtained using the high-resolution chip timer. Some benchmark code was inserted by the compiler (after modifications for benchmarking) and other code was inserted by hand to ensure accuracy. The benchmark programs are a number of Eiffel programs specifically designed to test the individual aspects being timed. A suite was constructed that ran each benchmark a number of times and recorded the best result.

benchmark	time (ms)
subsystem creation	57.40
request append	0.67
total signal	3.08
collect request	0.80

Table 2: Benchmark times for various aspects of the SCOOP system.

The first row in Table 2 shows the cost of subsystem creation. This involves creating a POSIX thread and allocating the memory required to represent the subsystem and processor (see Figure 6). The setup needed to create a new processor accounts for a large percentage of the time. Also, the system schedules the new thread to run immediately after its construction. The time reported here accounts for the time to setup and create the new processor, as well as for this processor to perform some of its own initialization, and, finally, enter a waiting state. This cost includes context switching times.

The second, third, and fourth rows of Table 2 measure part of the cost of calling a routine in another subsystem. Row two shows the time taken for a processor to append a new request to another processor's request queue. Row three shows the total time taken to attach a new request and signal the associated processor. The time taken to attach a new request is constant and does not depend on the length of the request queue. Row four shows the time taken for a processor, after finishing a routine, to check the request queue and start executing the next routine in the queue.

Locking

Mutual exclusion (locking) is obviously a central part of the system. The locking protocol provides the entry point through which concurrent routines are accessed. Figure 13 shows the cost of the locking algorithm as the time taken to lock and unlock a set of uncontested objects. The unlock time does not include the time taken to perform the HANDOFF-protocol, which depends on the number of waiting objects and their resource requirements.

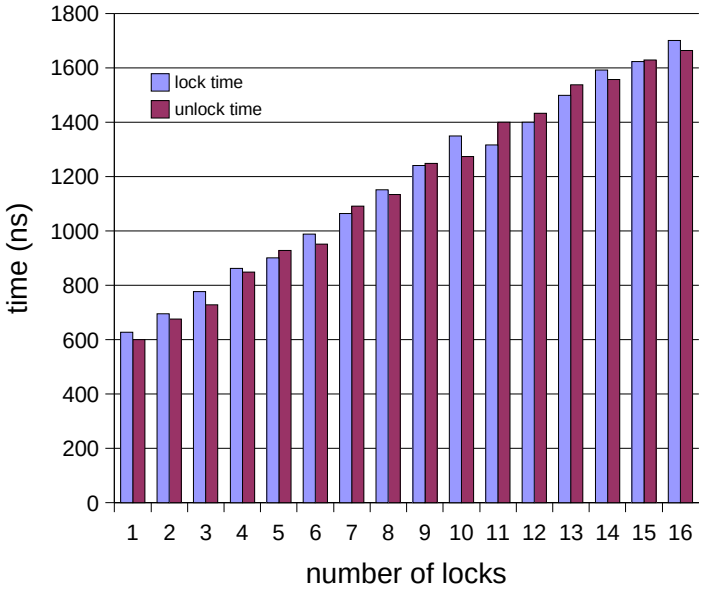


Figure 13: The time required to lock and unlock a set of uncontested locks (machine A).

Investigating Enhancements

Calling routines in different subsystems is one of the major concurrency features of SCOOP, which integrates inter-process communication (IPC) and feature access. Times are now presented to indicate the cost of calling a routine across a subsystem boundary (a concurrent routine). The benchmarks break down individual elements of the algorithm.

The separation provided by the calling mechanism means that the client of the call must also marshal the arguments and the routine and attach this structure to the request list of the client. Figure 14 demonstrates that the cost of marshalling arguments increases as the size of the data increases.

We now discuss the results of an experiment conducted into the reduction of this cost. The dominant cost of calling a separate routine is in allocating the necessary data structures (shown as the `malloc` line in Figure 14). A way to reduce this cost is to cache the structures and then reclaim them from the cache on later calls.

Figure 15 presents the results of an experiment using a caching mechanism to reduce the cost of calling routines across subsystems. The bar labelled *pack times* indicates the original pack time from Figure 14. This graph represents the average calling cost (for size 8–72 bytes). The average is presented here, as the cost of calling concurrent routines is not dominated by the size of the argument list. In fact, it has little weight in the total cost.

The bar labelled *global cache* is the result of an experiment to store the structures in a global cache accessible to all threads. Before marshalling arguments, a thread attempts to gain a pre-existing structure from the cache, and on completion of the call, the callee places the used structure back in the cache. It is expected that by using such an approach,

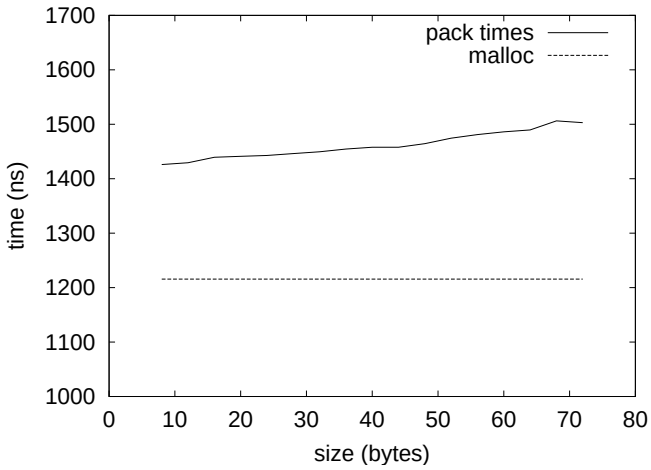


Figure 14: Time to marshal arguments for a call. It was found that this cost is dominated by the cost of allocating the data structures.

the cost of the `malloc` could be avoided in most cases. The results show that this represents a significant improvement, reducing the pack cost to under half the original cost.

A global cache is, however, a structure shared by all threads, and thus exclusive access to it is required; the cache is protected by a mutex. This makes the cache the subject of contention if two or more threads attempt to access it simultaneously. The bar labelled *global cache contention* shows that the time required to access the global cache increased under contention. In this experiment, four threads make repeated attempts to access the cache and pack data structures. The increase in time here is less than might be expected, but can be attributed to the fact that each thread holds the mutex for a very short period of time compared to the total call cost. It is not possible to conclude what level of contention would exist in a real program.

The final experiment in Figure 15 shows a best possible performance if each thread maintained its own cache. In this experiment each thread held a cache accessible through a thread-specific data key. As this cache does not require a mutex and is not the subject of contention, this cost would remain constant. It is unclear, however, if such an option could be supported properly in the system (unlike the global cache) and this result is provided for comparison only.

Although providing a cache for argument structures gives a significant performance increase, there are issues to be considered with such an option. The major issue is memory management. If these caches were to become large, they could waste memory better used for objects. A simple solution may be to limit the size of each cache. Another option may be for a garbage collector to reduce the size of these caches on garbage collection runs.

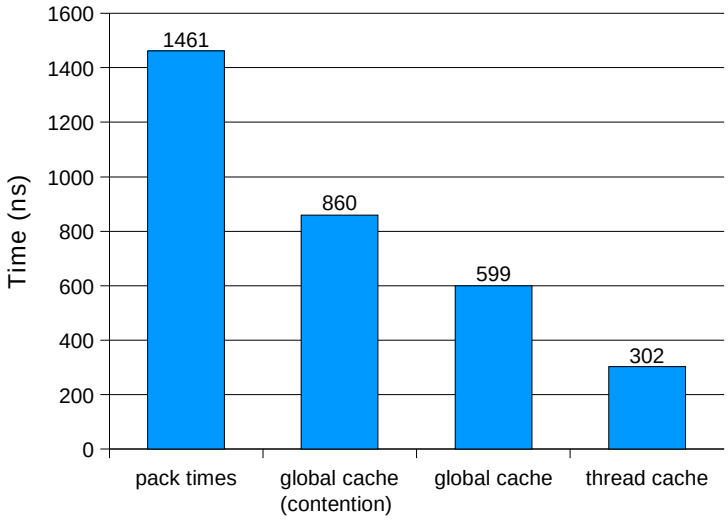


Figure 15: The results of an experiment to reduce the cost of calling routines in SCOOP. This graph presents the time taken to marshal a set of arguments under the different methods. The method of caching structures for marshalled arguments improves performance significantly.

Macro-benchmark

We now present a benchmark that shows performance of the run-time system as a whole. This benchmark demonstrates that for certain styles of program, the system can execute efficiently in parallel.

A program was constructed using Knuth's method to estimate the number of Eulerian cycles in a completely connected graph. The algorithm works by constructing a graph and running a number of trials to produce an estimate; these trials can be run on multiple threads concurrently. The algorithm suits the SCOOP model as there is no shared data between threads and the communication needs of threads are quite low.

SmallEiffel has three compilation modes:

- Normal mode compilation, which includes all require and ensure code checks (pre- and postconditions), as well as run-time type checking and bound checking on arrays;
- No_check mode, which removes assertion checking from the compiled code; and
- Boost mode, which removes all assertion checking and run-time type checking from compiled code. Boost mode also performs a number of optimizations, including the removal of SmallEiffel's run-time stack. The run-time stack records local variables and the original values of parameters passed to functions, partly for producing a stack trace of programs that violate assertions.

Figures 16–19 present the performance of the system as the number of threads increases. Each of these benchmarks was run on lightly loaded-machines and the size of the

problem was held constant (9 node graph with 10000 trials). Performance results for the three SmallEiffel compilation modes are shown.

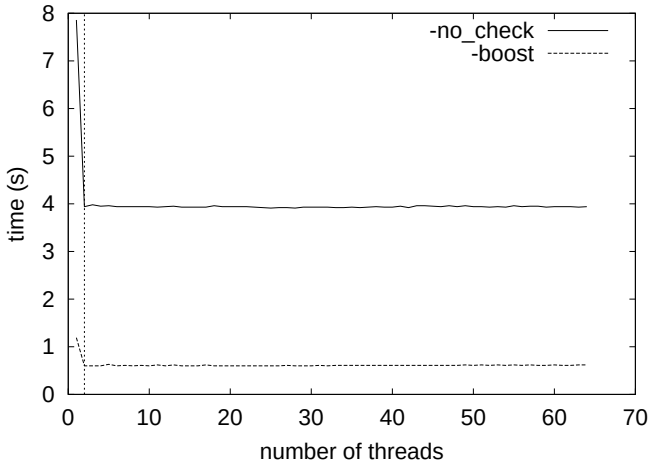


Figure 16: Machine *B*, a two-CPU Linux machine, shows a performance increase (for no_check and boost modes) until the point where no extra real parallelism can be gained.

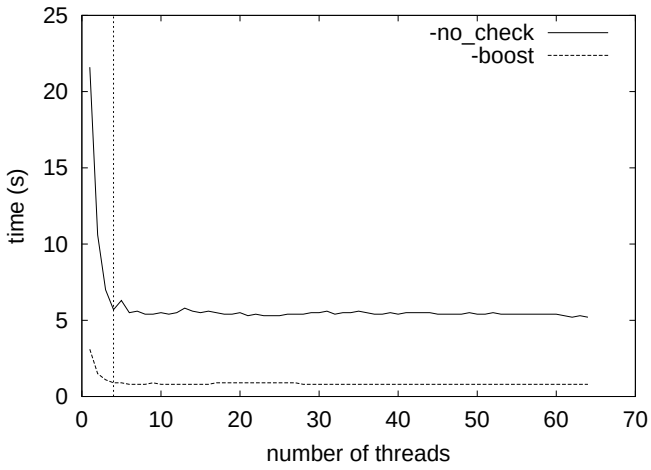


Figure 17: Times for no_check and boost modes on a four-CPU machine (machine *C*) decrease until the machine's CPUs are saturated with threads.

These graphs demonstrate that for an application that involves low communication, the system can produce impressive performance increases. For the two SMP machines (Figures 16 and 17) in the optimized modes, performance increased for each new processor that was used for the problem. This performance levelled off once no new parallelism could be gained from multiple CPUs.

Figures 18 and 19 show an interesting result. For normal mode, all three benchmark machines show performance improvements even after the machine's CPUs have been

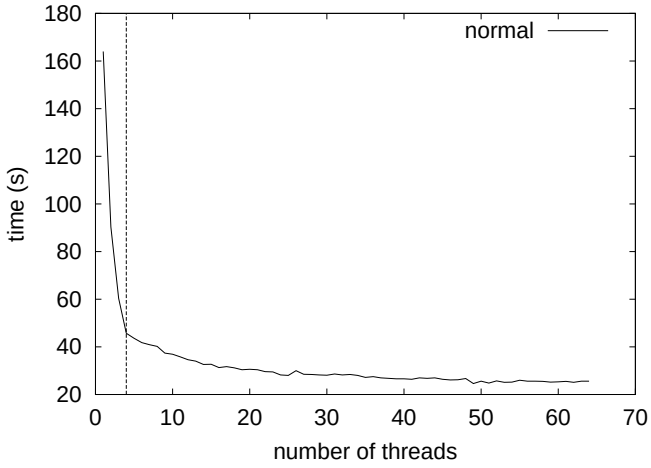


Figure 18: Results for machine *C* show that performance continues to increase after the machine's CPUs are saturated with threads.

saturated with threads. A good explanation for this has not been determined. It is, however, conjectured that this result could be the combination of two effects. First, a number of the overheads present in the unoptimized SmallEiffel run-time system could be being masked by the effects of multi-threading. Second, the performance increase could be the result of scheduling. It is possible that as the number of threads increases, there is an increase in the probability that a thread ready to perform some useful unit of computation is scheduled.

Figure 20 presents a finer-grained benchmark. This benchmark demonstrates that as the size of computation decreases in comparison to the amount of communication required between threads, there is a performance penalty associated with saturating the machine with threads. This example shows a small increase in communication overhead as the number of threads increases. A program that requires a greater amount of thread communication (and thus synchronization) would have a much greater overhead.

6 CONCLUSIONS AND FUTURE WORK

In the preceding sections we presented our initial implementation of SCOOP. In this section we indicate what remains to be considered and draw some conclusions.

Future Work

The semantics of Eiffel's exception mechanism in a concurrent context is not discussed in OOSC and needs careful thought. Consider a case where processing had continued well beyond the point of a separate call, perhaps even exiting the routine in which the call was made. An asynchronous exception at this time could not possibly have the same semantics as in the usual case.

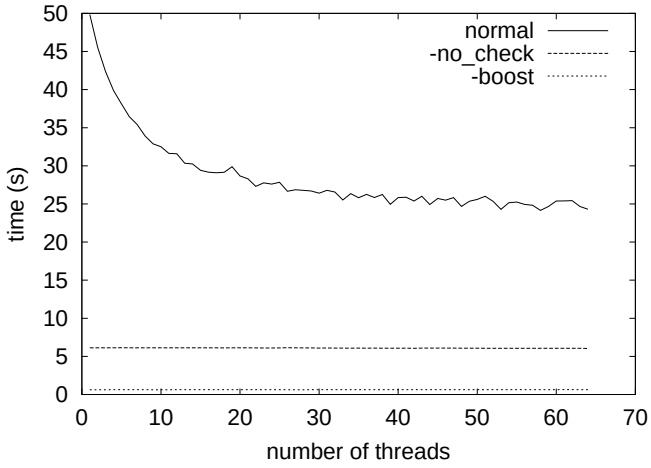


Figure 19: Performance of the Knuth benchmark on machine *A*.

We discussed this with Bertrand Meyer. There seem to be two possible solutions. Either, that an exception between subsystems is considered as catastrophic and causes program termination, or that the object throwing the exception is flagged as *dirty* and any later attempt to use this object results in program termination. We have adopted the first approach, but note that further investigation may lead to a better way.

An extension proposed by Meyer is a library, CONCURRENCY, which includes mechanisms to steal locks, request an express service, and yield execution in favour of other processors. OOSC does not provide a complete definition of the CONCURRENCY class and its features were not considered in depth during this project. It was considered best to investigate the central elements of the mechanism thoroughly before expanding to encompass aspects auxiliary to the workings of SCOOP.

Jalloul and Potter [11] have also discussed modifications of the SCOOP proposal, but we have not considered them in this project.

The definition of Eiffel contains a Makefile-like minilanguage, known as ACE, for specifying files to be compiled and compilation options. Meyer proposes a similar file for specifying the concurrency requirements of a SCOOP system. This new file is the Concurrency Control File (CCF). The SCOOP system constructed in this work does not use a CCF file. Instead, the concurrency requirements of a system are interpreted dynamically (at run time). The CCF file does, however, provide some interesting options if a distributed version of SCOOP were to be considered.

Concurrent garbage collection is a topic broad enough to warrant a project of its own for proper investigation. Again, there is no definite semantics for garbage collection in the context of SCOOP. Caromel [5] notes that a semantics where subsystems may become disconnected (unreachable) and still continue to process without garbage collection is a practical option.

The lack of a garbage collector certainly does not render the SCOOP compiler con-

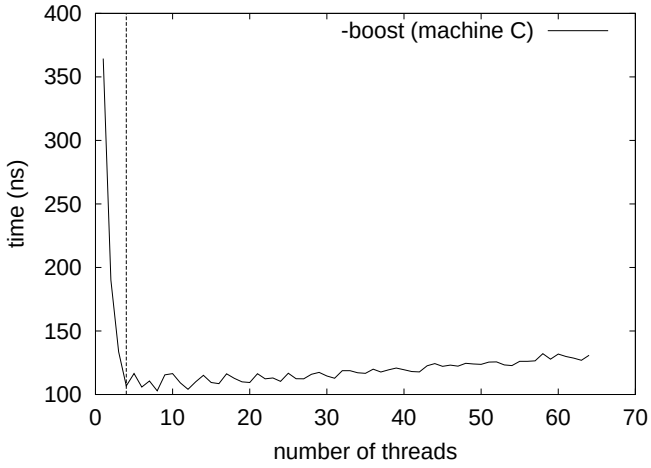


Figure 20: For fine-grained examples the system pays a performance penalty as the number of threads increases. This benchmark was conducted on machine *C*.

structed for this project unusable. Indeed, the SmallEiffel compiler existed for three years before a garbage collector was created for it. Many substantial programs including the compiler itself were written using SmallEiffel during this time. However, making garbage collection work must be considered an essential part of integrating fully the SCOOP mechanism into SmallEiffel.

We have also not considered alternative locking algorithms. Although this area is one of the most mature areas of computer science, it has remained remarkably active in recent years. New developments may provide further enhancements in the performance of the run-time system. The lock manager we have used does not allow different threads to lock non-intersecting sets of resources concurrently. An obvious extension of the run-time system is to implement an existing (or new) algorithm that allows this greater degree of concurrency. The trade-off between the concurrency allowed by such an algorithm and the communication it would require indicates that it may not perform better than ours.

Recent research in Java systems has shown that a review of existing programs can provide information useful in optimising a run-time system [3]. This indicates that revisiting the SCOOP run-time system, once a stock of representative SCOOP programs have been constructed and evaluated, could provide insights into the types of synchronisation, routine calling and contention that occur in real SCOOP programs.

And finally, the model we have presented brought some deficiencies in the definition of SCOOP to the surface; some decisions relating to the semantics remain to be made.

Concluding Remarks

We have defined a model that describes the SCOOP mechanism, and used it to derive the properties required of a run-time system. We have analysed all the requirements of SCOOP

and identified the problems needing to be addressed by an implementation. Our implementation of such a system provides an important step for Eiffel and is an improvement on previous efforts in this area. Our implementation uses POSIX libraries and is portable across a variety of operating systems.

We have presented benchmarks that demonstrate, in a way similar to the work of Nebro [20], that run-time systems for concurrent object-oriented languages can be implemented efficiently using standard threading models.

This is the first research project to consider SCOOP and the development of a compiler and run-time system needed to support it. As such, there is much more to explore. In particular, the availability of an implementation provides an opportunity to develop application programs that may help determine if SCOOP has the right level of expressiveness and power.

WHERE TO GET IT

Our prototype implementation, based on SmallEiffel release −0.76 beta 3, is available at `http://cs.anu.edu.au/people/Richard.Walker/eiffel/scoop/`. We welcome feedback.

ACKNOWLEDGEMENTS

We are grateful to Bertrand Meyer and Dominique Colnet for valuable discussions, advice, and encouragement. We also thank Malcolm Newey for his comments.

REFERENCES

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippe, Y Ramakrishna, and Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 207–222. ACM, 1999.

[2] Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. An Operational Semantics for the Eiffel// Language. Research Report 2732, Institut National de Recherche en Informatique et en Automatique (INRIA), 1995.

[3] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the SIGPAN'98 Conference on Programming Language Design and Implementation*, pages 258–268. ACM, June 1998.

[4] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[5] Dennis Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90—102, September 1993.

[6] K. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[7] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type inference for late binding. The SmallEiffel compiler. In *Proceedings of the Joint Modular Languages Conference 1997 (*JMLC*'97)*, volume 1204 of *Lecture Notes in Computer Science*, pages 67–81, 1997.

[8] Michael James Compton. SCOOP: An investigation of concurrency in Eiffel. Honours thesis, Department of Computer Science, The Australian National University, December 2000. Available at `http://cs.anu.edu.au/people/Richard.Walker/eiffel/scoop/`.

[9] Ian Foster and Stephen Taylor. A Compiler Approach to Scalable Concurrent-Program Design. *ACM Transactions on Programming Languages and Systems*, 16 (3):577–604, May 1994.

[10] International Organization for Standardization. Information technology – Portable Operating System Interface (POSIX®) – Part 1: System Application Program Interface (API) [C Language]. ISO·IEC 9945-1: 1996, ISO Geneva, 1996. Also available as IEEE·ANSI Standard 1003.1.

[11] Ghinwa Jalloul and John Potter. Models for concurrent Eiffel. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 6*, pages 183–191. Prentice Hall, 1991.

[12] Vijay Karamcheti and Andrew Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing'93*, 1993.

[13] Patrick Keane and Mark Moir. A general resource allocation synchronization problem. In *Proceedings of the 21st International Conference on Distributed Computing Systems (*ICDCS*-21)*, pages 557–564, April 2001.

[14] Andreas Krall and Mark Probst. Monitors and exceptions: how to implement Java efficiently. *Concurrency: Practice and Experience*, 10(11–13):837–850, 1998. Proceedings of the ACM 1998 Workshop on Java for High-performance Network Computing.

[15] Erwin Laure, Matthew Haines, Piyush Mehtrotra, and Hans Zima. On the Implementation of the Opus Coordination Language. *Parallel Processing Letters*, 9(2): 275–289, June 1999.

[16] Bertrand Meyer. Sequential and concurrent object-oriented programming. In *Proceedings of* TOOLS *'90*, pages 17–28, Paris, June 1990. Angkor/SOL.

[17] Bertrand Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice Hall, Hemel Hempstead, Hertfordshire, 1992.

[18] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the* ACM, 36(9):56–80, September 1993.

[19] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, New Jersey, second edition, 1997.

[20] Antonio Nebro, Ernesto Pimentel, and José Troya. Evaluating a Multithreaded Runtime System for Concurrent Object-Oriented Languages. *Proceedings of Computing in Object-Oriented Parallel Environments, Second International Symposium, ISCOPE 98 (Lecture Notes in Computer Science)*, 1505:167–174, December 1998.

[21] Injong Rhee. A modular algorithm for resource allocation. *Distributed Computing*, 11:157–168, 1998.

[22] Miguel Oliveira e Silva. Thread-safe SmallEiffel compiler. Available at http://wwsympa.loria.fr/wwsympa/arc/smalleiffel/2000-08/msg00034.html, August 2000. Last accessed 12 March 2002.

[23] Standard Performance Evaluation Corporation SPEC. SPEC JVM98 Benchmarks. http://www.spec.org/, 2000.

[24] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables. The SmallEiffel compiler. SIGPLAN *Notices*, 32 (10):125–141, October 1997. Proceedings of the 12th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97).

ABOUT THE AUTHORS

Michael Compton is a software engineer at CSIRO Mathematical and Information Sciences in Canberra. E-mail: Michael.Compton@csiro.au.

Richard Walker is an associate lecturer in the Department of Computer Science, Faculty of Engineering and Information Technology, at The Australian National University in Canberra. E-mail: Richard.Walker@cs.anu.edu.au.