Processors and their collection

Bertrand Meyer[1,2,3], Alexander Kogtenkov[2, 3], Anton Akhi[3]

[1]ETH Zurich, Switzerland
[3]Eiffel Software, Santa Barbara, California
[2]ITMO National Research University, Saint Petersburg, Russia

se.ethz.ch, eiffel.com, sel.ifmo.ru

Abstract. In a flexible approach to concurrent computation, "processors" ' (computational resources such as threads) are allocated dynamically, just as objects are; but then, just as objects, they can become unused, leading to performance degradation or worse. We generalized the notion of garbage collection (GC), traditionally applied to objects, so that it also handles collecting unused processors.

The paper describes the processor collection problem, formalizes it as a set of fixpoint equations, introduces the resulting objects-and-processor GC algorithm implemented as part of concurrency support (the SCOOP model) in the latest version of EiffelStudio, and presents benchmarks results showing that the new technique introduces no overhead as compared to traditional objects-only GC, and in fact improves its execution time slightly in some cases.

To appear in: MSEPT 2012, Prague, May 2012, eds. V. Pankratius & M. Philippsen, Springer LNCS, 2012.

1 Overview

Few issues are more pressing today, in the entire field of information technology, than providing a safe and convenient way to program concurrent architectures. The SCOOP approach to concurrent computation [5] [6] [7] [8] [9], devised in its basic form as a small extension to Eiffel, is a comprehensive effort to make concurrent programming as understandable and reliable as traditional sequential programming. The model, whose basic ideas go back to the nineties, has been continuously refined and now benefits from a solid implementation integrated in the EiffelStudio environment.

One of the starting ideas of SCOOP, which it shares with some other concurrency models, is the notion of a "processor" as the basic unit of concurrency. A processor is a mechanism that can execute a sequence of instructions; it can concretely be implemented in many ways, either in hardware as a CPU, or in software as a single-threaded process or a thread.

When processors are implemented in software, they get created in a way similar to objects in object-oriented programming and, like objects, they may become inactive, raising a problem of garbage collection (GC). While object GC has been extensively studied (see [4] for a recent survey), we are not aware of previous discussions of processor GC, save for a discussion of a partly related problem for actors in [3]. What makes the problem delicate is that processor GC must be intricately connected with the classical *object* GC: to decide that a processor P is no longer useful and can be collected, it is not enough to ascertain that P has no instructions left to execute; we must also make

sure that no live object from another processor has a reference to an object handled by *P* and hence retains the possibility or soliciting *P*.

The present article discusses processor garbage collection as implemented in the latest release of EiffelStudio. It is not an introduction to SCOOP (which may be found in the references listed above) and indeed presents the concepts in a form that can be applied to many other concurrency models.

Section 2 explains the notion of processor as a general concurrency mechanism. Section 3 introduces the problem of collecting processors. Section 4 describes the constraints on any solution. Section 5 formalizes the problem as a set of two mutually recursive equations and introduces the resulting fixpoint algorithm. Section 6 presents the results of a number of benchmarks, showing no degradation and, in some case, performance improvements. Section 7 describes possibilities for future work.

The mechanism presented here has been fully implemented as part of the SCOOP implementation included in EiffelStudio version 7.1, available in both open-source and commercial licenses and downloadable from the Eiffel site [2].

2 Processors

The concept of processor captures the basic difference between sequential and concurrent computation: in sequential mode, there is only one mechanism capable of executing instructions one after the other; in concurrent mode, we may combine several such mechanisms. The problems of concurrency arise out of the need to coordinate these individual sequential computations.

2.1 Handlers and regions

This definition of processors would be close to a platitude — concurrent computation is concurrent because it can pursue several sequential tasks at once — were it not for the connection with object-oriented programming as established in SCOOP: the assignment of every *object* to a single processor. Object-oriented computation is dominated by the basic operation

```
x.f (args)
```

a *feature call* (also called “method call” and “message passing”), which applies a feature (operation) *f* to an object *x*, the **target** of the call, with some optional arguments *args*. Concurrent mechanisms that have not been specifically designed for the object-oriented paradigm, such as Java Threads, enforce no particular connection between the concurrency structure (the division into processors) and the object structure; the standard risks of concurrent programming, in particular data races, arise as a result, and can only be avoided through programmer discipline such as the use of “synchronized” calls. SCOOP makes the connection between the processor and object structures by assigning, for every object **O**, a single processor — the object’s **handler** — to execute all calls having **O** as their target. Since processors are sequential, this decision also means that at most one operation may be proceeding on any given target object at any

given time (although compiler optimization may produce exceptions to this rule as long as they preserve the semantics).

The practical consequence of this policy is a partition of the object space in one-to-one correspondence with the set of processors:

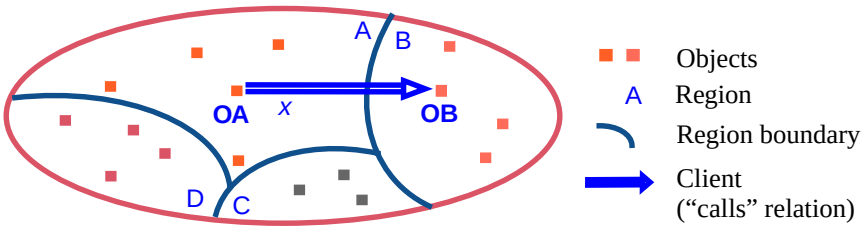

Fig. 1: Processors cause a partition of objects into regions

Each element of the partition, containing all the objects handled by a particular processor, is called a **region**.

In the figure, a call may be executed on behalf of the object **OA**, and hence as part of a computation performed by its handler, the processor A; since the call has the target *x* denoting the object **OB**, handled by another processor B, it will be executed by B.

A call handled by a different processor, as here since the call starts from A but is executed by B on its behalf, calls for appropriate semantics; specifically in SCOOP, the call is:

- **Asynchronous** if the feature is a command (a procedure, executing some actions), since there is no need in this case for A to wait; if it did, there would be no benefit from concurrency.
- **Synchronous** if it is a query (a function computing a result, or a lookup of a value in a field) since in this case the result is needed for the computation to proceed. This is the mechanism of wait by necessity [1] [5].

To make this specific behavior clear, programmers must declare **separate** (the only new Eiffel keyword for SCOOP) a variable such as *x* to specify that it may denote objects handled by a processor other than the current object's handler. Such a declaration does not specify the processor; it simply states that this processor may be different, and hence lead to a different semantics for calls. The SCOOP type system [8] guarantees semantic consistency; in particular it is not permitted to assign from separate to non-separate.

2.2 Call vs application

One of the consequences of treating command calls with a separate target as asynchronous is to introduce a refinement of the classical notion of feature call *x.f(args)*. In a concurrent context we must distinguish between the feature's *call* and its *application*:

- The calling processor (A in the earlier figure) executes a call. If the call is asynchronous, the processor does not wait and proceeds to its next instruction. At

the implementation level, the only needed operation is to **log** the call with the supplier processor, which typically uses a queue to record such logged requests).

- At some later time, the supplier processor (B in the figure) will be ready to execute the call's instruction. This is the actual **application** of the feature.

This separation between call and application is the defining property of asynchrony as permitted by concurrent computation.

2.3 Call logging

Every processor may log several separate calls to different processors. One way to implement processors is to set up each of them as a loop that looks at the queue of logged calls, retrieves one call, and applies it. It is not possible to determine automatically when to terminate this loop (any more than to solve any other general termination problem). We may note, however, that a processor is no longer useful when:

- No calls are logged on it (at the implementation level, its queue is empty).
- No object from another (live) processor contains a reference to one of its own objects.

Note that the second condition does not imply the first: even though the client processor A that executed *x*.*f* (*args*) had a reference — namely, *x* — to an object **OB** of the supplier processor B at the time of the call, A may not be keeping any direct interest in the result of the call; the operation that executed *x*.*f* (*args*) may itself have terminated and the object **OA** that contained it may have been reclaimed. It is still obligated, however, to execute the logged operation.

3 Collecting processors

We will now review the issues raised by extending traditional garbage collection to a concurrent environment.

3.1 The need for processor garbage collection

For general discussions of concurrency, and for writing concurrent programs, it does not matter how the processors are physically implemented; the general definition that a processor is a mechanism capable of executing instructions sequentially suffices. In the context of the present discussion a processor is a software mechanism; in the current EiffelStudio implementation, processors are indeed implemented as threads, although future versions may provide other representations, particularly in a distributed context.

Concretely, a processor gets created every time a creation instruction (equivalent to a "new" in C++/Java syntax) is executed with a separate target:

```
create x.make (...)    -- With x: separate T for some type T.
                       -- make is the creation procedure (constructor).
```

(Here too compilers optimizations may avoid the creation of a new physical resource, for example the implementation may reuse an existing thread, but conceptually the instruction creates a new processor.)

Since processors are allocated dynamically, the same problem arises as with objects in ordinary object-oriented programming languages supporting dynamic object allocation (such as Java, Eiffel or C#): some processors may become unused; if not

reclaimed, they will waste resources, and possibly lead to resource exhaustion and program freezing. For example an inactive thread takes up space to hold its local data, and takes up CPU time if the thread scheduler continues to examine it. As with objects, it may be desirable to reclaim — garbage-collect — unused processors.

3.2 Challenges of processor GC

Object garbage collection is a classic research topic with an abundant literature, which we will not attempt to review, referring the reader instead to [4]. To what extent can the concepts apply to processor GC?

Any garbage collection mechanism, whether for objects or for some other resource, conceptually includes two aspects: *detection*, which identifies unused resources, and *reclamation*, which frees them or recycles them for new needs. Typical schemes such as "mark-and-sweep" refine this idea: the mark phase implements detection by traversing the object structure, starting from a set of root objects known to be alive and recursively following all references to flag all the objects it reaches; the sweep phase implements reclamation by traversing all memory and reclaiming all unmarked objects (as well as unmarking all objects in preparation for the next GC cycle).

In trying to transpose these concepts to processor collection, the principal issue is that it is not sufficient, for detection, to determine that a processor has no more instructions of its own to execute: another condition, already noted in 2.3, is that *none of its objects has an incoming reference from an object handled by another processor*. Were such a reference to exist, it could later on cause a new request for computation if the other processor is itself (recursively) still active. This specification is sufficiently delicate to require a formal specification, to be given in section 5.3.

4 Practical requirements on an objects+processors GC

An effective solution to processor collection must take into account a number of practical issues.

4.1 Triggering conditions

In classical object GC, the trigger for a collection cycle is typically that memory usage has gone beyond a certain threshold.

The concurrent GC scheme adds another threshold, on the number of processors. If processors have been allocated beyond that threshold, the GC will be triggered to reclaim any unused processors.

4.2 Root objects

The need to take object references into account shows that the mechanisms of object GC and processor GC are not independent, but mutually recursive. In line with this observation, the algorithm that we have implemented integrates processor GC within the preexisting object GC algorithm, maintaining a queue of active processors initialized with the processors known to be active (the processors that still have instructions to execute) and enriching it, during object traversals, with the processors handling objects that are found to be reachable.

The concurrent setup introduces one more issue not present in sequential GC: root objects. All GC algorithms need to start from a set (the *root set*) of objects known for sure to be alive (the *root objects*). The first complication is that we must deal not with a single root set but with a multiplicity of root sets, one per processor. Another issue arises in the case of global objects; in the Eiffel context these are the result of "once functions" (functions that, as the name implies, are executed — "applied" in the earlier terminology — only once, upon their first call, with the result saved and returned in any subsequent call). In another language, static variables would raise a similar difficulty. If there are no references from other processors to the result of a once function, the basic algorithm just outlined would collect it; this behavior is clearly unsound, but the solution of treating all such objects as roots is also unsatisfactory as they would then never be collected even though some of them may not be live.

4.3 Memory overhead

Taking processors into account requires supplementary memory structures. In particular it is necessary to record a "processor ID" for any object, identifying the object's handler. Fortunately, we were able in EiffelStudio's internal representation of objectsto reuse two heretofore available bytes. As a consequence, the memory overhead is zero.

4.4 Time overhead

A straightforward extension to conventional object GC would handle every processor as if it were an object. Such a special processor object would contain references to the objects in its root set and, conversely, every object would have an implicit reference to the processor object corresponding to its processor ID.

This solution introduces an overhead since it adds a conceptual reference to every object. The benchmarks (see section 6.4) confirm that the overhead would be significant. The algorithm described below avoids it by separating the objects and the processors. In addition:

- The overhead during object traversal consists of a single unconditional write that can be efficiently handled by the out-of-order instruction execution available on today's CPUs.
- All the memory used to track live processors is allocated in a very small contiguous chunk that fits the CPU cache; this technique avoids cache misses and reduces the write time to the minimum.
- The most expensive traversal part associated with the root sets of the processors is executed separately.

The result of these optimizations, confirmed by the benchmarks of section 6, is that the implementation avoids any significant slowdown as compared to the non-concurrent GC collecting objects only, and in fact slightly improves the performance in some cases.

4.5 Object revival

Many GC-enabled languages and environments offer the possibility of associating with objects of a certain type a "finalization" routine (in Eiffel, *dispose* from the library class *DISPOSABLE*) which will be called whenever a GC cycle reaches one of these objects, say A, considered dead. In the absence of any restriction, such a mechanism threatens the soundness and completeness of the garbage collector:

- Although A has been marked as dead (ready for collection), the finalization routine could add a reference to A from some other object B that is live, reviving A and preventing its collection.
- The routine could execute a call *x*.*f* (...) using a reference *x* in A, but the corresponding object C might be dead.

In a concurrent setting the referenced objects could have a different handler, so finalization could cause the revival of a processor.

Because of these problems, the Eiffel environment enforces a strong restriction on finalization routines, both in a sequential setting and in SCOOP: such a routine may not include any qualified call (that is to say, any call *x*.*f* (...) with an explicit target *x*). Unqualified calls (*f* (...), applying to the current object) are permitted.

4.6 Partial GC

Modern "generational" object GC systems support partial garbage collection, which reclaims some objects without traversing the entire heap. The EiffelStudio implementation performs frequent partial collection, which minimizes the performance impact on the computation, and occasional full collection, to reclaim any dead objects that the partial GC cycles did not detect.

As will be detailed below, object GC in a concurrent context mutually depends on processor GC. We have not yet found a way to integrate this double GC mechanism in the partial collection algorithm. As a consequence, processor GC only occurs during the full collection cycles.

5 Devising an objects+processors GC problem

We now describe the GC design, starting with an informal description of the problem and continuing with a mathematical description and a presentation of the algorithm.

5.1 Root sets

As noted in 4.2, the starting point of any GC process is the root set. In a concurrent setting the root set contains two parts:

- A set of *system-wide root objects*, not related to any processor.
- For each live processor, a set of *processor-specific root objects*.

The precise definition of liveness for processors appears next (5.2).

The second part, processor-specific root objects, includes for each processor: objects on its call stack; objects on its evaluation stack (when the implementation uses interpreted code); objects on other run-time stacks (in the presence of calls to external software, as supported for example by Eiffel's C/C++ interface); results of processor-level once functions; activation records containing the targets and arguments of separate feature calls.

As specified, we need only consider the processor-specific root object sets of *live* processors. This property causes a modification of the object GC algorithm: instead of starting from all potential root objects it can restrict itself to live processors.

5.2 When is a processor ready for collection?

To determine when processors are “live” and “dead”, we note that the typical steps in the life of a processor are the following:

- On processor creation, logging a call to a creation procedure (constructor, such as *make* in the instruction **create** *x*.*make* (...) where *x* is separate, which creates a new object on a new processor and initializes the object through *make*).
- As a result of a call from another processor, logging a separate feature call.
- If the log queue contains a call ready for application, removing it from the queue and applying the feature.

A processor is dead when it cannot perform any useful work, right now or in the future. This is the case when both:

- It has no currently logged calls.
- No calls can ever be logged in the future.

The first condition is local to the processor; the second one involves the entire system. This second condition, however, is undecidable. We need a stronger condition that can be checked; that condition is that the processor’s objects are not reachable through references from live objects (from any processors). It is clearly stronger than needed, since we do not know that such references will ever be followed, but it is sound. Hence the definition of liveness that we retain for practical purposes:

Definition: dead, live processor

A processor is **dead** if both:

1 It has no calls logged and not yet applied.

2 None of its objects is referenced by a live object.

A processor is **live** if it is not dead.

The set of live objects, necessary for the second part of the definition, is obtained by traversal of the object structure starting from the root set. We have just seen, however, that the root set includes the processor-specific root sets of live processors. As a consequence, the definitions of liveness for objects and processors are mutually recursive; they will now be formalized.

5.3 Formal description

We may describe the processor collection object mathematically as follows. The two sets of interest are *LO*, the set of live objects, and *LP*, the set of live processors. They will be defined by a set of two mutually recursive equations.

We assume a set *BP* (for “basic processors”) of processors known to be live — as their log queues are not empty — and a set *BR* (“basic roots”) of objects known to be live.

The function *h* (for “handler”) maps objects to their processors. We will write *h* (*o*) not only when *o* denotes a single object but also when it denotes a set of objects, the

result then being a set of processors. In other words we use the same name *h* for the handler function and the associated image function.

For a processor *p*, *r* (*p*) (*r* for "roots") denotes the set of its root objects. As in the previous case, *r* will also be applied to sets of processors; *r* (*P*), for a set *P* of processors, denotes the union of the individual root sets of the processors in *P*.

The set of objects to which an object *o* contains references (links) is written *s* (*o*) (*s* for "successors"), again generalized to sets of objects. As usual, * denotes reflexive transitive closure, so that the set of objects reachable from the objects in a set *O* is *s** (*O*).

The sets *LO* and *LP* of live objects and processors depend on each other and on the reference structure, as defined by the following equations:

LO	=	$s^*(BR \cup r(LP))$	/1/
LP	=	$BP \cup h(LO)$	/2/

5.4 Algorithm

/1/ and /2/ is a fixpoint equation of the form $f = \tau(f)$ on functions *f* applying to [*LO*, *LP*] pairs. We are looking for a minimum fixpoint (with respect to the partial order defined by set inclusion, generalized to [*objects*, *processors*] subset pairs), since we should only retain objects and processors that are strictly necessary, and reclaim any others. The function τ is monotonic; since the underlying sets of objects and processors are finite, fixpoint theory tells us that a minimal fixpoint exists and can be obtained as the finitely reached limit of the sequences LO_i and LP_i defined as follows:

LO_0	=	BR	/3/
LP_0	=	BP	/4/
LO_{i+1}	=	$s(LO_i \cup r(LP_i))$	/5/
LP_{i+1}	=	$LP_i \cup h(LO_i)$	/6/

which readily yields the basic algorithm:

```
from                                                          /7/
        LO := BR ; LP := BP                  -- done initialized to False
until
        done
loop
        saved_LO := LO
        LO := s (LO ∪ r (LP))
        LP := LP ∪ h (saved_LO)
        done := "No change to LO and LP since last iteration"
end
```

The algorithm is guaranteed to terminate as a consequence of the preceding observations. In practice we can do away with *saved_LO* since the algorithm remains sound if we replace the body of the loop by just

LO := *s* (*LO* ∪ *r* (*LP*))
LP := *LP* ∪ *h* (*LO*)
done := "No change to *LO* and *LP* since last iteration"

The algorithm can be further improved by computing at each step the difference between the new and old values of LO and LP, rather than recomputing the whole sets each time. /7/ with these two improvements (for the details of the implementation, the reader can refer to the open-source code available from [2]) is the basic algorithm for combined object-processor collection, as has been implemented in EiffelStudio 7.1.

6 Performance evaluation

To evaluate the performance, we wrote three test programs that allocate up to 10 million objects and use up to 100 processors. Calculations have been performed on a computer with a 3.2GHz AMD Phenom II processor and 4GB of RAM.

The first test involves independent data structures, each local to a processor; the next ones use structures that are distributed among processors.

6.1 Test setup

Each test proceeded through the following procedure, repeated fifty times with the results then averaged:

- Turn off garbage collection (through the corresponding mechanisms in the Eiffel libraries).
- Create object structures.
- Explicitly trigger a full garbage collection (again through a library mechanism).

6.2 Test results

The first test is intended to assess the basic overhead of adding processor GC to object GC, in the case of independent data structures. It creates full binary tree structures, with various heights, on different processors. All the nodes in each tree have the same handler, so that garbage collection could be performed almost independently on different processors.

Figure 2 shows the time dependency on number of processors for two tree depths. Numbers of objects in this figure and the following ones are in thousands.

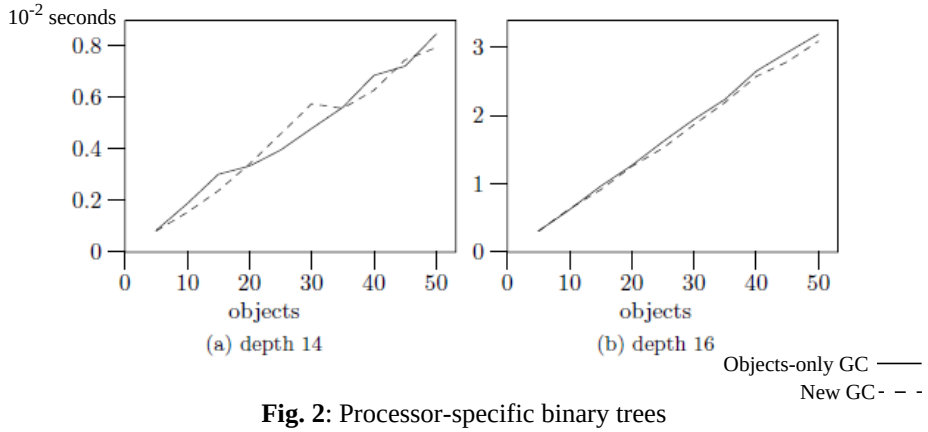

Fig. 2: Processor-specific binary trees

The second test consists of cyclic structures, with nodes randomly allocated to processors; it is intended to measure the GC algorithm's ability to move from processor to processor during the marking phase, and to collect structures with cyclic links. Figure 3 shows the time dependency on the number of processors ((a) and (b)) and objects ((c) and (d).

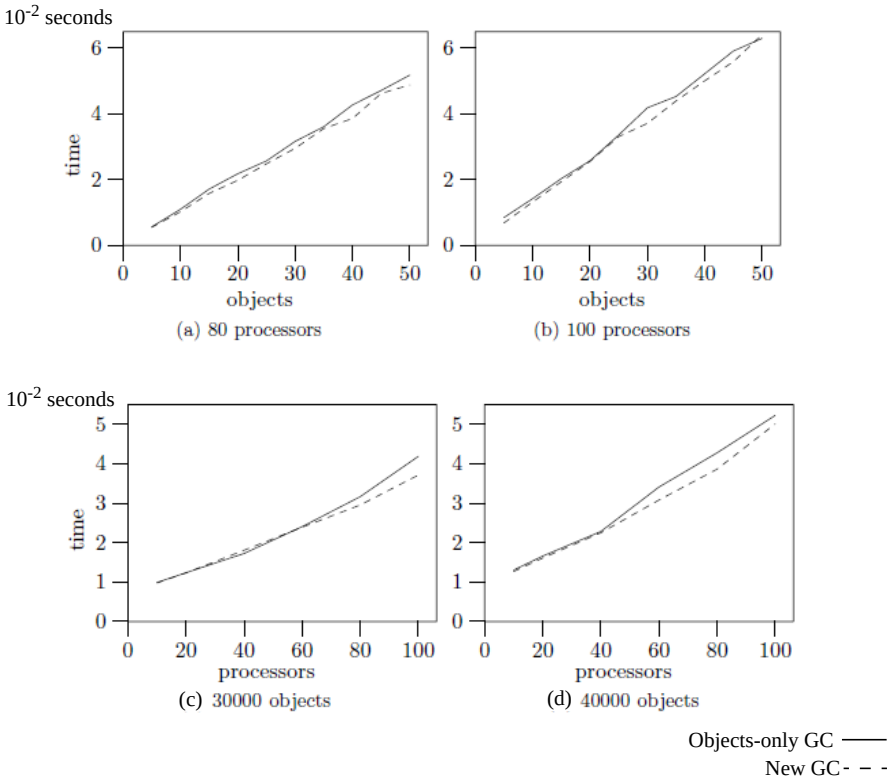

Fig. 3: Randomly distributed cyclic structures

The next test randomly creates objects on processors. From each object, up to ten random links to other objects were created. Because of the large number of objects, their spread across processors, and the large number of links, this test yields many interesting cases for the algorithm. Figure 4 shows the outcome.

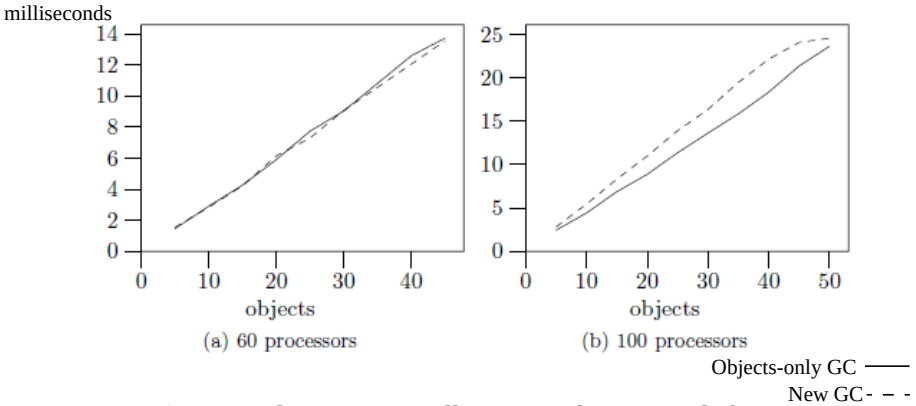

Fig. 4: Random processor allocation and numerous links

6.3 Assessment

The results shown above indicate that in general the new algorithm's execution time remains essentially the same as that of the previous, objects-only GC.

In some cases the new algorithm is actually faster. Several factors may explain this improvement:

- The new algorithm does not need to perform marking of objects on dead processors.
- Since it only traverses objects from live processors, there is a good chance that some of those objects were recently used by program and are available in the processor's cache. The previous algorithm could cause loads from memory in such cases.
- The new algorithm visits objects in a different order, which may have some effects on the performance.

6.4 Assessing the processor-as-object approach

We added a test to compare the proposed algorithm and the straightforward approach of treating processors simply as objects, discussed in section 4.4. The test emulates the resulting overhead by adding a reference field to every object. Figure 5 shows the comparison with the algorithm without such reference fields.

For practical reasons, the reference field has been added in both versions; it is simply void (null) in the "new GC" version. Although this technique introduces a small difference with the real algorithm, we believe that any resulting bias is very small (and probably to the detriment of the new algorithm).

The results of the test clearly show an overhead of 6% to 12% for the processors-as-objects approach. This overhead linearly increases with the number of objects, whereas there is no noticeable increase with the retained algorithm.

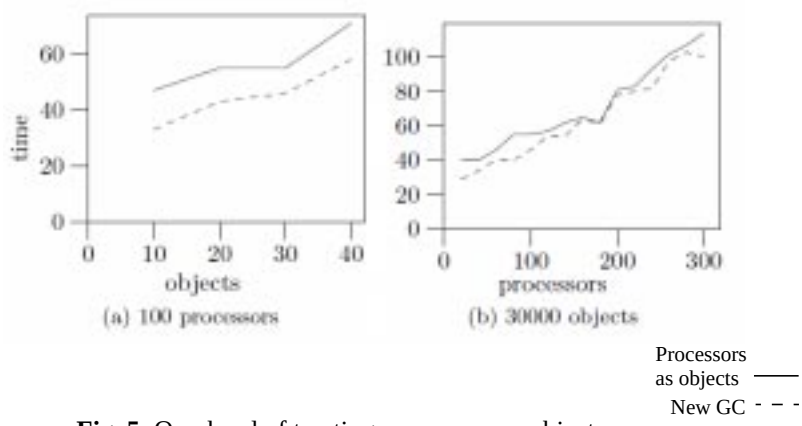

Fig. 5: Overhead of treating processors as objects

7 Other applications and future work

As noted, the problem address in this article, garbage-collecting processors, is essentially new. The work reported here has shown that an efficient and sound solution is possible. The key idea is to treat object GC and processor GC as intricately (and recursively) connected, modeling and performing them together.

While implemented for Eiffel and SCOOP, the algorithm relies only on the properties listed in this article, in particular the notion of processor; it is independent of many characteristics of a programming language, such as its type system, and is therefore of potential application to different models of concurrency.

Some of the highlights of the approach are that:

- Memory is not shared but distributed among execution resources.
- The algorithm makes it possible to provide information about active resources.
- At presents it only works in the context of a full GC cycles.
- There is a reachability function for memory resources.

The last requirement does not assume fine-grained resolution: it is sufficient to work on the level of memory regions belonging to the specific execution flow. The approach discussed here does use object-level information to collect unused processors, but only to demonstrate that this scheme can be naturally integrated with the existing traditional GC. This leads us to assume that the algorithm can be applied in a completely distributed setting, but we have not yet examined this extension (including support for fault tolerance) in detail.

An important topic for further research is automatic management of reclaimed execution resources. In this paper we intentionally left out the details about acquiring and releasing resources from the underlying operating environment. In some systems where these operations are costly, it may make sense to preallocate pools of execution resources and apply load balancing, to allow efficient operation in highly dynamic conditions.

Bibliography

[1] Denis Caromel: Towards A Method of Object-Oriented Concurrent Programming, in Communications of the ACM, vol. 36, no. 9, September 1993, pages 90-102.

[2] EiffelStudio environment, available for download at eiffel.com.

[3] Dennis Kafura, Doug Washabaugh and Jeff Nelson: *Garbage collection of actors*, in OOPSLA/ECOOP '90, 1990, pages 126-134.

[4] Richard Jones, Antony Hosking and Eliot Moss: *The Garbage Collection Handbook*: *The Art of Automatic Memory Management*, Chapman and All/CRC, 2nd edition, 2011.

[5] Bertrand Meyer: *Systematic Concurrent Object-Oriented Programming*, in *Communications of the ACM*, vol. 36, no. 9, September 1993, pp. 56-80.

[6] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997 (chapter 32 presents SCOOP).

[7] Benjamin Morandi, Sebastian Nanz and Bertrand Meyer: *A Formal Reference for SCOOP*, in *Empirical Software Engineering and Verification* (LASER 2008-2010), eds. B. Meyer and M. Nordio, Lecture Notes in Computer Science 7007, Springer-Verlag, 2012.

[8] Piotr Nienaltowski: *Practical framework for contract-based concurrent object-oriented programming*, PhD dissertation 17061, Department of Computer Science, ETH Zurich, February 2007, available at se.ethz.ch/old/people/nienaltowski/papers/thesis.pdf.

[9] Piotr Nienaltowski, Jonathan Ostroff and Bertrand Meyer: *Contracts for Concurrency*, in *Formal Aspects of Computing Journal*, vol. 21, no. 4, August 2009, pages 305-318.

[10] Paul R. Wilson: *Uniprocessor Garbage Collection Techniques*, in *Proceedings of the International Workshop on Memory Management* (IWMM '92), eds. Y. Bekkers and K. Cohen, Springer-Verlag, 1992, pages 1-42.

Acknowledgments

The implementation described here is part of the SCOOP mechanism of EiffelStudio, developed at Eiffel Software, to which key other contributors are Emmanuel Stapf and Ian King. The benefit of discussions with members of the SCOOP team at ETH Zurich, in particular Sebastian Nanz, Benjamin Morandi and Scott West, is gratefully acknowledged.

We are greatly indebted to the funding agencies that have made the work on SCOOP possible. The SCOOP project at ETH has been supported by grants from the Swiss National Science Foundation and the Hasler foundation, an ETH ETHIIRA grant, and a Multicore Award from Microsoft Research. The ITMO Software Engineering Laboratory is supported by a grant from Mail.ru group.