Safe and Efficient Data Sharing for Message-Passing Concurrency

Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer

Department of Computer Science, ETH Zurich, Switzerland,
firstname.lastname@inf.ethz.ch,
http://se.inf.ethz.ch/

Abstract. Message passing provides a powerful communication abstraction in both distributed and shared memory environments. It is particularly successful at preventing problems arising from shared state, such as data races, as it avoids sharing in general. Message passing is less effective when concurrent access to large amounts of data is needed, as the overhead of messaging may be prohibitive. In shared memory environments, this issue could be alleviated by supporting direct access to shared data; but then ensuring proper synchronization becomes again the dominant problem. This paper proposes a safe and efficient approach to data sharing in message-passing concurrency models based on the idea of distinguishing active and passive computational units. Passive units do not have execution capabilities but offer to active units exclusive and direct access to the data they encapsulate. The access is transparent due to a single primitive for both data access and message passing. By distinguishing active and passive units, no additional infrastructure for shared data is necessary. The concept is applied to SCOOP, an object-oriented concurrency model, where it reduces execution time by several orders of magnitude on data-intensive parallel programs.

1 Introduction

In concurrency models with message passing, such as the Actor model [14], CSP [15], and others [6], a *computational unit* encapsulates its own private data. The units interact by sending synchronous or asynchronous messages. These concurrency models are implementable in environments with and without shared memory. Based on these models, several languages and libraries support message passing, e.g., Erlang [8], Ada [16], MPI [19], and SCOOP [22].

To operate on shared data, a *client* must send a message to the *supplier* encapsulating that data. In environments with shared memory, however, the client could access this data directly and avoid the messaging overhead. The difficulty is to prevent data races and to combine the data access primitives with the messaging primitives in a developer-friendly way.

Some languages and libraries [8, 16, 19, 25] have already combined mutually exclusive shared data with message passing and observed performance gains on shared memory systems. However, as discussed in Section 6, these approaches

either impose restrictions on the shared data or do not provide unified primitives for data access and message passing. As a consequence of the latter limitation, programmers are required to change their code substantially when switching from messaging to shared data or vice versa.

To close this gap, this paper proposes the concept of *passive* computational units for safe and efficient data sharing in message-passing models implemented on shared memory. A passive unit is a supplier stripped from its execution capabilities. Its only purpose is to provide a container for shared data and exclusive access to it. The passive unit can contain any data that is containable by a regular supplier. A client with exclusive access uses existing communication primitives to operate on the data; instead of sending a message, these primitives access the data directly. By overloading the semantics of the primitives, programmers only need to change few lines of code to set a unit passive. Furthermore, passive units are implementable with little effort as existing supplier infrastructure can be reused.

This paper develops this concept in the context of SCOOP [20,22], an object-oriented concurrency model based on message passing, where *processors* encapsulate objects and interact by sending requests. The implementation of the concept is shown to reduce execution time by several orders of magnitude on data-intensive parallel programs.

The remainder of this paper is structured as follows. Section 2 introduces a running example. Section 3 develops the concept of *passive processors* informally, and Section 4 develops it formally. Section 5 evaluates the efficiency, and Section 6 reviews related work. Finally, Section 7 discusses the applicability to other concurrency models and concludes with an outlook on future work.

2 Pipeline System

A pipeline system serves as the running example for this paper. The pipeline parallel design pattern [18] applies whenever a computation involves sending *packages* of data through a sequence of *stages* that operate on the packages. The pattern assigns each stage to a different computational unit; the stages then synchronize with each other to process the packages in the correct order. Using this pattern, each stage can be mapped for instance to a CPU core, a GPU core, an FPGA, or a cryptographic accelerator, depending on the stage's computational needs.

The pipeline pattern can be implemented in SCOOP (Simple Concurrent Object-Oriented Programming) [20,22]. The starting idea of SCOOP is that every object is associated with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing actions on objects. An object's class describes its actions as *features*. An *entity* x belonging to a processor can point to an object with the same handler (*non-separate object*), or to an object on another processor (*separate object*). In the first case, a *feature call* $x.f$ on the *target* x is *non-separate*: the handler of x executes the feature synchronously. In the second case, the feature call is *separate*: the handler of x, i.e., the *supplier*,

executes the call asynchronously on behalf of the requester, i.e., the *client*. The possibility of asynchronous calls is the main source of concurrent execution. The asynchronous nature of separate feature calls implies a distinction between a feature call and a *feature application*: the client logs the call with the supplier (feature call) and moves on; only at some later time will the supplier actually execute the body (feature application).

In the SCOOP pipeline implementation, each stage and each package is handled by its own processor, ensuring that stages can access the packages in parallel. Each stage is numbered to indicate its position in the pipeline; it receives this position upon creation:

```
class STAGE create make feature
  position: INTEGER -- The stage's position in the pipeline.

  make (new_position: INTEGER)
      -- Create a stage at the given position.
    do position := new_position end

  process (package: separate PACKAGE)
      -- Process the package after the previous stage is done with it.
    require package.is_processed (position - 1) do
      do_work (package) -- Read from and write to the package.
      package.set_processed (position) -- Set the package processed.
    end
end
```

The *process* feature takes a package as an argument; the keyword **separate** specifies that the package may be handled by a different processor. To ensure exclusive access, a stage must first lock a package before accessing it. In SCOOP, such locking requirements are expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. In *process*, *package* is a formal argument; hence the stage has exclusive access to the package while executing *process*.

To process each package in the right order, the stages must synchronize with each other. For this purpose, each package has two features *is_processed* and *set_processed* to keep track of the stages that already processed the package. The synchronization requirement can then be expressed elegantly using a precondition (keyword **require**), which makes the execution of a feature wait until the condition is true. The precondition in *process* delays the execution until the package has been processed by the previous stage.

The SCOOP concepts require execution-time support, known as the SCOOP runtime. Each processor is protected through a lock and maintains a *request queue* of requests resulting from feature calls of other processors. When a client executes a separate feature call, it enqueues a *separate feature request* to the supplier's request queue. The supplier processes the feature requests in the order of queuing. A non-separate feature call can be processed without the request

queue: the processor creates a *non-separate feature request* and processes it right away using its call stack.

A client makes sure that it holds the locks on all suppliers before executing a feature. At the end of the feature execution, the client issues an unlock request to each locked processor. Each locked processor unlocks itself as soon as it processed all previous feature requests.

3 Passive Processors

The pipeline system from Section 2 showcases an important class of concurrent programs, namely those that involve multiple processors sharing data. In SCOOP, it is necessary to assign the data to a new processor. For frequent and short read or write operations this becomes problematic:

1. Each feature call to the data leads to a feature request in the request queue of the data processor, which then picks up the request and processes it on its call stack. This chain of actions creates a considerable overhead. For an asynchronous write operation, the overhead outweighs the benefit of asynchrony. For a synchronous read operation, the client not only waits for the data processor to process the request, it also gets delayed further by the overhead.
2. The data consumes operating system resources (threads, processes, locks, semaphores) that could otherwise be freed up.

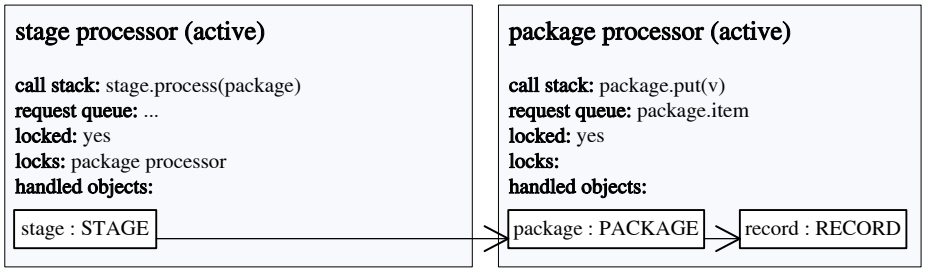
On systems with shared memory, the clients can directly operate on the data, thus avoiding the overhead. This frees most of the operating system resources attached to the data processor. Before accessing shared data, a client must ensure its access is mutually exclusive; otherwise, data races can occur. For this purpose, shared data must be grouped, and each group must be protected through a lock. Since SCOOP processors offer this functionality already along with execution capabilities, one can use processors, stripped from their execution capabilities, to group and protect shared data. This insight gives rise to passive processors:

Definition 1 (Passive processor). *A* passive *processor q does not have any execution capabilities. Its lock protects the access to its associated objects. A client p holding this lock uses feature calls to operate directly on q's associated objects. While operating on these objects, p assumes the identity of q. Processor q becomes passive when another processor sets it as passive. When q is not passive, it is* active. *Processor q becomes active again when another processor sets it as active. It can only become passive or active when unlocked, i.e., when not being used by any other processor.*

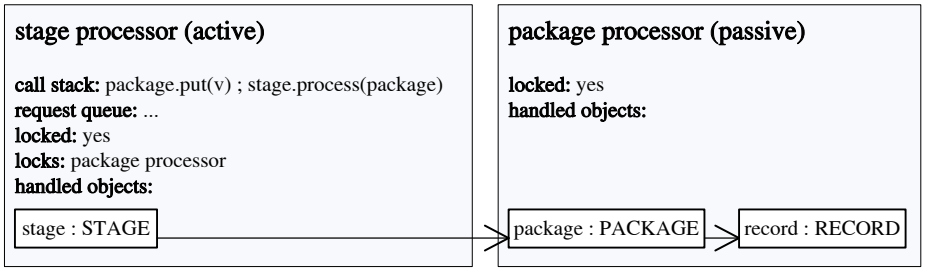
When a processor p operates on the objects of a passive processor q, it assumes q's identity. For example, if p creates a literal object or another non-separate object, it creates this object on q and not on itself; otherwise, a non-separate entity on q would reference an object on p.

Besides safe and fast data sharing, passive processors have further benefits:

– *Minimal user code changes.* The feature call primitive unifies sending messages to active processors and accessing shared data on passive processors, ensuring minimal code changes to set a processor passive or active. With respect to SCOOP's type system [22], the same types can be used to type objects on passive and active processors. The existing type system rules ensure that no object on a passive processor can be seen as non-separate on a different processor, thus providing type soundness.
– *Minimal compiler and runtime changes.* To implement passive processors, much of the existing infrastructure can be reused. In particular, no new code for grouping objects and for locking request queues is required.



(a) The package processors are active.



(b) The package processors are passive.

Fig. 1: A stage processor processes a package. The stage object, handled by the left-hand side processor, has a separate reference (depicted by an arrow) to the package object, handled by the right-hand side processor. The package object references a non-separate record object to remember the processing history.

In the pipeline system, each package can be handled by a passive processor rather than an active one. To achieve this, it suffices to set a package's processor passive after its construction. No other code changes are necessary since the existing feature calls to the packages automatically assume the data access semantics. For example, the following code creates a package on a new passive processor and asks the stages to process the package.

create *package.make* (*data*, *number_of_stages*) ; *set_passive* (*package*)
stage_1.process (*package*) ; ... ; *stage_n.process* (*package*)

Figure 1 illustrates the effect of the call to *set_passive*. In Figure 1a, active package processors have a call stack, a request queue, and a stack of locks. The stage processors send asynchronous (see *put*) and synchronous (see *item*) feature requests. In Figure 1b, passive package processors do not have any execution capabilities. Therefore, the stage processors operate directly and synchronously on the packages, thus making a better use of their own processing capabilities rather than relaying all operations to the package processors.

4 Formal Specification

This section provides a structural operational semantics for the passive processor mechanism and shows that setting a processor passive or active preserves the execution order of called features.

4.1 State Formalization

Let Ref be the type of references, let $Proc$ be the type of processors, and let $Entity$ be the type of entities. A state σ is then a 6-tuple $(\sigma_h, \sigma_l, \sigma_o, \sigma_i, \sigma_f, \sigma_e)$ of functions:

- $\sigma_h : Ref \to Proc$ maps each reference to its handler.
- $\sigma_l : Proc \to Boolean$ indicates which processors are locked.
- $\sigma_o : Proc \to Stack[Set[Proc]]$ maps each processor to its obtained locks.
- $\sigma_i : Proc \to Boolean$ indicates which processors are passive.
- $\sigma_f : Proc \to Proc$ maps each processor p to the handler of the object on which p currently operates. Normally $\sigma_f(p) = p$, but when operating on the objects of a passive supplier q, then $\sigma_f(p) = q$.
- $\sigma_e : Proc \to Stack[Map[Entity, Ref]]$ maps each processor to its stack of entity environments.

4.2 Execution Formalization

An *execution* is a sequence of configurations. Each *configuration* of the form $\langle p_1 :: s_{p1} \mid \ldots \mid p_n :: s_{pn}, \sigma \rangle$ is an execution snapshot consisting of the *schedule*, i.e., the call stacks and the request queues of processors $p_1, \ldots, p_n$, and the state σ. The call stack and the request queue of a processor are also known as the *action queue* of the processor. The commutative and associative parallel operator $\mid$ keeps the processors' action queues apart. Within an action queue, a semicolon separates statements. *Statements* are either *instructions*, i.e., program elements, or *operations*, i.e., runtime elements. The following overview shows the structure of statements, instructions, and operations. The elements e_t, e, and all items in $\overline{e_a}$ are entities of type $Entity$. The element r_t and all items in $\overline{r_a}$ are references of type Ref. The element f of type $Feature$ denotes a feature where $f.body$

returns the feature's body. Lastly, $q_1, \ldots, q_n$ and w are processors of type *Proc*, and x is a flag of type *Boolean*.

$s \triangleq in \mid op$

$in \triangleq$

$e_t.f(\overline{e_a})$ |	Call a feature.
create $e_t.f(\overline{e_a})$ |	Create an object.
set_passive(e) |	Set the handler of the referenced object passive.
set_active(e)	Set the handler of the referenced object active.

$op \triangleq$

`apply`$(r_t, f, \overline{r_a})$ |	Apply a feature.
`revert`$(\{q_1, \ldots, q_n\}, w, x)$ |	Finish a feature application or an object creation.
`unlock`	Unlock a processor.

Figure 2 shows the transition rules. A processor q becomes passive when a processor p executes the *set_passive* instruction (see Set Passive) with an entity e that evaluates to a reference r on q. Processor q becomes active again when a processor p executes the *set_active* instruction (see Set Active). Processor q can only become passive or active when q is unlocked, guaranteeing that q is not being used by any other processor.

To perform a feature call $e_t.f(\overline{e_a})$ (see call rules) a client p evaluates the target (see r_t) and the arguments (see $\overline{r_a}$). It then looks at the handler q of the target. If q is different from p and not passive, p creates a feature request (see `apply`) and appends it to the end of q's request queue. If q is p or if q is passive, then p itself immediately processes the feature request.

To process a feature request (see Apply), a processor p first determines the missing locks $\overline{q}$ as the difference between required locks and already obtained locks. It only proceeds when all missing locks are available, in which case it obtains these locks. It also adds a new entity environment and updates σ_f with the target handler, i.e., p for non-passive calls or the handler of the target for passive calls. Processor p then executes the feature body and cleans up (see Revert). It releases the obtained locks, restores σ_f, and removes the top entity environment. The locked suppliers unlock themselves asynchronously once they are done with the issued workload (see Unlock).

To execute a creation instruction **create** $e_t.f(\overline{e_a})$ (see creation rules), a processor p looks at the type of the target e_t. If the type is separate, i.e., its declaration has the **separate** keyword, p creates an active and idle processor q with a new object referenced by r_t. It then locks that processor, performs the creation call (see call rules), and cleans up. If the type of e_t is non-separate, i.e., no **separate** keyword, p creates a new object on the handler on whose objects p currently operates on, i.e., $\sigma_f(p)$. In case $\sigma_f(p) = q \neq p$, it is important to create the new object on q rather than on p; otherwise the non-separate entity e_t on q would point to an object not on q, thus compromising the soundness of the type system.

We embedded the transition rules from Figure 2 into the comprehensive formal specification for SCOOP [10], implemented in Maude [5]. This specification

SET PASSIVE

$$\frac{r \stackrel{def}{=} \sigma_e(p).top.val(e) \;\; q \stackrel{def}{=} \sigma_h(r) \;\; \neg\sigma_l(q)}{\langle p :: set\_passive(e); s_p, \sigma\rangle \to \langle p :: s_p, (\sigma_h, \sigma_l, \sigma_o, \sigma_i[q \mapsto true], \sigma_f, \sigma_e)\rangle}$$

SET ACTIVE

$$\frac{r \stackrel{def}{=} \sigma_e(p).top.val(e) \;\; q \stackrel{def}{=} \sigma_h(r) \;\; \neg\sigma_l(q)}{\langle p :: set\_active(e); s_p, \sigma\rangle \to \langle p :: s_p, (\sigma_h, \sigma_l, \sigma_o, \sigma_i[q \mapsto false], \sigma_f, \sigma_e)\rangle}$$

SEPARATE CALL

$$\frac{r_t \stackrel{def}{=} \sigma_e(p).top.val(e_t) \;\; \overline{r_a} \stackrel{def}{=} \sigma_e(p).top.val(\overline{e_a}) \;\; q = \sigma_h(r_t) \;\; p \neq q \wedge \neg\sigma_i(q)}{\langle p :: e_t.f(\overline{e_a}); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: s_p \mid q :: s_q; \texttt{apply}(r_t, f, \overline{r_a}), \sigma\rangle}$$

NON-SEPARATE/PASSIVE CALL

$$\frac{r_t \stackrel{def}{=} \sigma_e(p).top.val(e_t) \;\; \overline{r_a} \stackrel{def}{=} \sigma_e(p).top.val(\overline{e_a}) \;\; q = \sigma_h(r_t) \;\; p = q \vee \sigma_i(q)}{\langle p :: e_t.f(\overline{e_a}); s_p, \sigma\rangle \to \langle p :: \texttt{apply}(r_t, f, \overline{r_a}); s_p, \sigma\rangle}$$

APPLY

$$\frac{\overline{q} \stackrel{def}{=} \sigma_h(\overline{r_a}) \setminus (\sigma_o(p).flat \cup \{p\}) \;\; \bigwedge_{q\in\overline{q}} \neg\sigma_l(q)}{\begin{array}{l}\langle p :: \texttt{apply}(r_t, f, \overline{r_a}); s_p, \sigma\rangle \to \\ \langle p :: f.body; \texttt{revert}(\overline{q}, \sigma_f(p), true); s_p, (\sigma_h, \sigma_l[\overline{q} \mapsto true], \sigma_o[p \mapsto \sigma_o(p).push(\overline{q})], \sigma_i, \\ \qquad \sigma_f[p \mapsto \sigma_h(r_t)], \sigma_e[p \mapsto \sigma_e(p).push((current \mapsto r_t, f.formals \mapsto \overline{r_a}))]\rangle\end{array}}$$

REVERT

$$e' \stackrel{def}{=} \begin{cases} \sigma_e[p \mapsto \sigma_e(p).pop] & if\, x \\ \sigma_e & otherwise \end{cases}$$

$$\frac{}{\begin{array}{l}\langle p :: \texttt{revert}(\{q_1, \ldots, q_n\}, w, x); s_p \mid q_1 :: s_{q1} \mid \ldots \mid q_n :: s_{qn}, \sigma\rangle \to \\ \langle p :: s_p \mid q_1 :: s_{q1}; \texttt{unlock} \mid \ldots \mid q_n :: \ldots, (\sigma_h, \sigma_l, \sigma_o[p \mapsto \sigma_o(p).pop], \sigma_i, \sigma_f[p \mapsto w], e')\rangle\end{array}}$$

UNLOCK

$$\frac{}{\begin{array}{l}\langle p :: \texttt{unlock}; s_p, \sigma\rangle \to \\ \langle p :: s_p, (\sigma_h, \sigma_l[p \mapsto false], \sigma_o, \sigma_i, \sigma_f, \sigma_e)\rangle\end{array}}$$

PARALLELISM

$$\frac{\langle P, \sigma\rangle \to \langle P', \sigma'\rangle}{\langle P \mid Q, \sigma\rangle \to \langle P' \mid Q, \sigma'\rangle}$$

SEPARATE CREATION

$$\frac{e_t.type = separate \;\; q \stackrel{def}{=} fresh\_proc(\sigma_h) \;\; r_t \stackrel{def}{=} fresh\_obj(\sigma_h)}{\begin{array}{l}\langle p :: \textbf{create}\ e_t.f(\overline{e_a}); s_p, \sigma\rangle \to \langle p :: e_t.f(\overline{e_a}); \texttt{revert}(\{q\}, \sigma_f(p), false); s_p \mid q ::, \\ \qquad (\sigma_h[r_t \mapsto q], \sigma_l[q \mapsto true], \sigma_o[p \mapsto \sigma_o(p).push(\{q\})][q \mapsto ()], \sigma_i[q \mapsto false], \\ \qquad\qquad \sigma_f[q \mapsto q], \sigma_e[p \mapsto \sigma_e(p).update(e_t \mapsto r_t)][q \mapsto ()])\rangle\end{array}}$$

NON-SEPARATE CREATION

$$\frac{e_t.type = non\text{–}separate \;\; r_t \stackrel{def}{=} fresh\_obj(\sigma_h)}{\begin{array}{l}\langle p :: \textbf{create}\ e_t.f(\overline{e_a}); s_p, \sigma\rangle \to \\ \langle p :: e_t.f(\overline{e_a}); s_p, (\sigma_h[r_t \mapsto \sigma_f(p)], \sigma_l, \sigma_o, \sigma_i, \sigma_f, \sigma_e[p \mapsto \sigma_e(p).update(e_t \mapsto r_t)])\rangle\end{array}}$$

Fig. 2: Transition rules

uses $\sigma_f(p)$ also in other situations where p performs an action on behalf of a passive processor, namely to create literals, to set the status of a once routine (a routine only executed once), and to import and copy object structures. We used the specification to test [21] the passive processor mechanism against other SCOOP aspects and used the results to refine the specification.

4.3 Order Preservation

The formal semantics can be used to prove Theorem 1, stating that a supplier can always be set passive or active without altering the sequence in which called features get applied. This property enables developers to use the same reasoning in determining a feature's functional correctness, irrespective of whether the suppliers are passive or active. Lemma 1 is necessary to prove Theorem 1:

Lemma 1 (Action queue order preservation). *Let p be a processor with statements $s_1, \ldots, s_l$ in its action queue. In a terminating program, p will execute $s_1, \ldots, s_l$ in the sequence order.*

Proof. The transition rules in Figure 2 only allow p to execute the leftmost statement and then continue with the next one. Since none of the rules delete or shuffle any statements, and since p's program is terminating, p must execute $s_1, \ldots, s_l$ in the sequence order.

Theorem 1 (Feature call order preservation). *Let p be a processor that is about to apply a feature f in a terminating program. Let $\overline{q}$ be the processors that p locks to apply f. For each $q \in \overline{q}$, regardless whether it is passive or active, the feature requests for q, resulting from feature calls in f's body, will be processed in the order given by f's body.*

Proof. Processor p first inserts f's instructions $s_1, \ldots, s_l$ into its action queue (see Apply). Lemma 1 states that processor p executes all of these instructions in code order. Hence, the proof can use mathematical induction over the length l of f's body. In the base case, i.e., $l = 0$, p did not execute any instructions; hence, the property holds trivially. For the inductive step, the property holds for $l = i - 1$; the proof needs to show that the property holds for $l = i$. Consider the instruction s_i at position i:

- s_i is a *set_passive* or *set_active* instruction. Processor p does not change any action queues (see Set Passive and Set Active); the property is preserved.
- s_i is a separate feature call to a passive processor q. Processor p executes the feature call (see Non-Separate/Passive Call). Processor q must already have been passive during earlier calls because a processor cannot be set passive when it is locked (see Apply and Set Passive). Hence, processor p must have processed all earlier calls. Because of the induction hypothesis, it must have done so in the order given by the code. Consequently, processing s_i now preserves the property for q. Because of the induction hypothesis,

the configuration after s_{i-1} satisfied the property for all other suppliers in $\overline{q}$; Lemma 1 guarantees that these processors will execute their statements in the same order even after s_i, thus the property is preserved.

- s_i is a separate feature call to an active processor q. Processor p executes a separate call (see SEPARATE CALL) to add a feature request to the end of q's action queue. Because of the induction hypothesis, q must either have processed all earlier calls in the code order, or some of these calls must be scheduled in q's action queue, to be executed in code order. In either case, adding a feature request for s_i to the end of the action queue preserves the property for q (see Lemma 1). As in the passive case, Lemma 1 guarantees that the property is preserved for all other suppliers in $\overline{q}$ as well.
- s_i is a non-separate feature call. Regardless of the suppliers' passiveness, p executes a non-separate feature call (see NON-SEPARATE/PASSIVE CALL). Lemma 1 guarantees that the property is preserved for all q in $\overline{q}$.
- s_i is a separate creation instruction. Regardless of the suppliers' passiveness, p executes a separate feature call (see SEPARATE CREATION), adding a new feature request to a new processor. Lemma 1 guarantees that the property is preserved for all q in $\overline{q}$.
- s_i is a non-separate creation instruction. Regardless of the supplier's passiveness, p executes a non-separate feature call (see NON-SEPARATE CREATION). Lemma 1 guarantees that the property is preserved for all q in $\overline{q}$.

5 Evaluation

The pipeline system from Section 2 is a good representative for the class of programs targeted by the proposed mechanism: multiple stages share packages of data. This section experimentally compares the performance of the pipeline system when implemented using passive processors, active processors, and low-level synchronization primitives; the latter two are the closest competing approaches. To this end, we extended the SCOOP implementation [9] with passive processors.

5.1 Comparison to active processors

A low-pass filter pipeline is especially suited because it exhibits frequent and short read and write operations on the packages, each of which represents a signal to be filtered. The pipeline has three stages: the first performs a decimation-in-time radix-2 fast Fourier transformation [17]; the second applies a low-pass filter in Fourier space; and the third inverses the transformation. The system supports any number of pipelines operating in parallel and splits the signals evenly.

Table 1 shows the average execution times of various low-pass filter systems processing signals of various lengths. The experiments have been conducted on a 4 × Intel Xeon E7-4830 2.13 GHz server (32 cores) with 256 GB of RAM running Windows Server 2012 Datacenter (64 Bit) in a Kernel-based Virtual Machine on Red Hat 4.4.7-3 (64 Bit). A modified version of EVE 13.11 [9] compiled the programs in finalized mode with an inline depth of 100. Every data point reflects

the average execution time over ten runs processing 100 signals each. Using ten pipelines, it took nearly ten hours to compute the average for active signals of length 16384; thus we refrained from computing data points for bigger lengths.

Table 1: Average execution times (in seconds) of various low-pass filter systems with various signal lengths.

configuration	2048	4096	8192	16384	32768	65536	131072	262144	524288
sequential, SCOOP	1.00	1.66	3.22	6.35	12.71	26.19	55.05	120.37	272.38
sequential, thread	0.62	1.09	2.19	4.66	9.57	20.23	41.45	93.40	213.59
1 pipeline, active	337.55	682.29	1456.67	2875.23	-	-	-	-	-
1 pipeline, passive	1.64	2.72	5.02	10.99	24.68	55.29	118.30	247.29	533.83
1 pipeline, thread	0.31	0.58	1.16	2.44	5.26	11.11	23.59	53.24	122.00
2 pipelines, passive	1.19	1.77	3.05	6.35	14.19	29.82	60.24	124.99	263.96
3 pipelines, passive	1.07	1.47	2.34	4.71	9.87	20.65	41.78	85.72	185.19
5 pipelines, active	231.25	496.68	1048.95	2192.53	-	-	-	-	-
5 pipelines, passive	0.87	1.23	1.86	3.27	6.35	13.50	27.17	55.95	117.12
5 pipelines, thread	0.16	0.23	0.40	0.74	1.53	3.05	6.30	13.76	31.82
10 pipelines, active	334.93	726.83	1549.01	3322.70	-	-	-	-	-
10 pipelines, passive	0.84	1.08	1.38	2.36	4.13	8.28	16.89	35.13	76.64
10 pipelines, thread	0.16	0.22	0.33	0.59	1.08	2.09	4.26	9.21	20.93

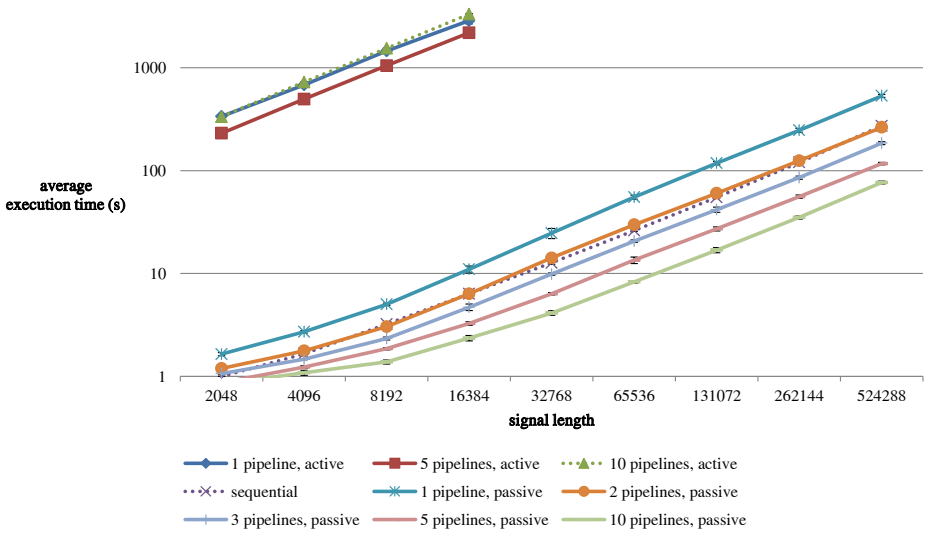


Fig. 3: The speedup of passive processors over active processors

Figure 3 visualizes the data. The upper three curves belong to the active signal processors. The lower curves result from the passive processors and a sequential execution. As the graph indicates, the passive processors are more than two orders of magnitude faster than the active ones. In addition, with increasing number of pipelines, the passive processors become faster than the

sequential program. In fact, two pipelines are enough to have an equivalent performance. The overhead is thus small enough to benefit from an increase in parallelism. In contrast, active processors deliver their peak performance with around five pipelines but never get faster than the sequential programs.

5.2 Comparison to low-level synchronization primitives

Figure 4 and Table 1 compare pipelines with passive processors to pipelines based on low-level synchronization primitives. In the measured range, the passive processors are between 3.7 to 5.4 times slower. As the signal length increases, the slowdown tends to becomes smaller. With more pipelines, the slowdown also tends to decrease at signal lengths above 8192. The two curves for sequential executions show that a slowdown can also be observed for non-concurrent programs.

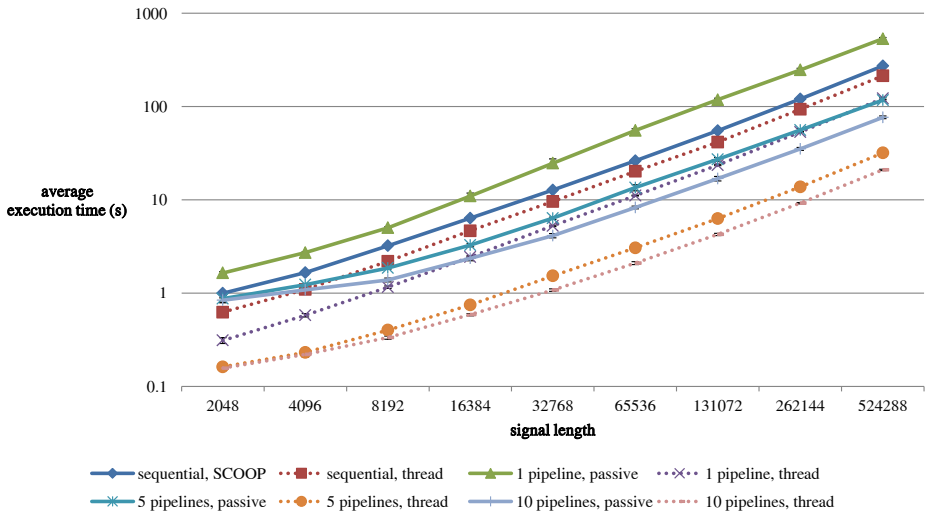


Fig. 4: The slowdown of passive processors over EiffelThread

The slowdown is the consequence of SCOOP's programming abstractions. Compare the following thread-based stage implementation to the SCOOP one from Section 2. Besides the addition of boilerplate (inherit clause, redefinition of *execute*), this code exhibits some more momentous differences. First, the thread-based stage class implements a work queue: it has an attribute to hold the packages and a loop in *execute* to go over them. In SCOOP, request queues provide this functionality. Second, each thread-based package has a mutex and a condition variable for synchronization. To process a package, stage i first locks the mutex and then uses the condition variable to wait until stage $i-1$ has processed the package. Once stage $i-1$ is done, it uses the condition variable to

signal all waiting stages. Only stage i leaves the loop. In SCOOP, wait conditions provide this kind of synchronization off-the-shelf. We expect the cost of wait conditions and other SCOOP concepts to drop further, as the compiler and the runtime mature.

```
class STAGE inherit THREAD create make feature
  position: INTEGER -- The stage's position in the pipeline.
  packages: ARRAY[PACKAGE] -- The packages to be processed.

  make (new_position: INTEGER; new_packages: ARRAY[PACKAGE])
      -- Create a stage at the given position to operate on the packages.
    do position := new_position ; packages := new_packages end

  execute
      -- Process each package after the previous stage is done with it.
    do
      across packages as package loop
        package.mutex.lock -- Lock the package.
        -- Sleep until previous stage is done; release the lock meanwhile.
        from until package.is_processed (position - 1) loop
          package.condition_variable.wait (package.mutex)
        end
        process (package) -- Process the package.
        package.condition_variable.broadcast -- Wake up next stage.
        package.mutex.unlock -- Unlock the package.
      end
    end

  process (package: PACKAGE)
      -- Process the package.
    do
      do_work (package) -- Read from and write to the package
      package.set_processed (position) -- Set the package processed.
    end
end
```

5.3 Other applications

A variety of other applications could also profit from passive processors. Object structures can be distributed over passive processors. Multiple clients can thus operate on dynamically changing but distinct parts of these structures while exchanging coordination messages. For example, in parallel graph algorithms, the vertices can be distributed over passive processors. In producer-consumer programs, intermediate buffers can be passive. Normally, about half of the operations in producer-consumer programs are synchronous read accesses. Without the messaging overhead, the consumer can execute these operations much faster than the buffer. Passive processors can also be useful to handle objects whose only purpose it is to be lockable, e.g., forks of dining philosophers, or to encapsulate a shared state, e.g., a robot's state in a controller.

6 Related work

Several languages and libraries combine shared data with message passing. In Ada [16], *tasks* execute concurrently and communicate during a *rendezvous*: upon joining a rendezvous, the client waits for a message from the supplier, and the supplier synchronously sends a message to the client. The client joins a rendezvous by calling a supplier's *entry*. The supplier joins by calling *accept* on that entry. To share data, tasks access *protected objects* that encapsulate data and provide exclusive access thereon through guarded functions, procedures, and entries. Since functions may only read data, multiple function calls may be active simultaneously. In contrast, passive processors do not support multiple readers. However, unlike protected objects, passive processors do not require new data access primitives. Furthermore, passive processors can become active at runtime.

Erlang [8] is a functional programming language whose concurrency support is based on the actor model [14]. Processes exchange messages and share data using an *ets table*, providing atomic and isolated access to table entries. A process can also use the *Mnesia* database management system to group a series of table operations into an atomic transaction. While passive processors do not provide support for transactions, they are not restricted to tables.

Schill et al. [25] developed a library offering indexed arrays that can be accessed concurrently by multiple SCOOP processors. To prevent data races on an array, each processor must reserve a *slice* of the array. Slices support fine-grained sharing as well as multiple readers using *views*, but they are restricted to indexed containers. For instance, distributed graphs cannot be easily expressed.

A group of MPI [19] processes can share data using the *remote memory access* mechanism and its *passive target* communication. The processes collectively create a *window* of shared memory. Processes access a window during an *epoch*, which begins with a collective synchronization call, continues with communication calls, and ends with another synchronization call. Synchronization includes fencing and locking. Locks can be partial or full, and they can be shared or exclusive. Passive processors neither offer fences nor shared locks; they do, however, offer automatic conditional synchronization based on preconditions. MPI can also be combined with OpenMP [23]. Just like MPI alone, this combination does not provide unified concepts. Instead, it provides distinct primitives to access shared data and to send messages. Uniformity also distinguishes passive processors from further approaches such as [13].

Several studies agree that performance gains can be realized if the setup of a program with both message passing and shared data fits the underlying architecture. For instance, Bull et al. [3] and Rabenseifner et al. [24] focus on benchmarks for MPI+OpenMP. Wei and Yilmaz [27] study an adaptive integral method to analyze electromagnetic scattering from perfectly conducting surfaces. Dunn and Meyer [7] use a QR factorization algorithm that can be adjusted to apply only message passing, only shared data, or both.

A number of approaches focus on optimizing messaging on shared memory systems instead of combining message passing with shared data. Gruber and Boyer [12] use an ownership management system to avoid copying messages be-

tween actors while retaining memory isolation. When an actor sends a message, the system transfers ownership over the message to the receiver. From that point on, only the receiver can access the message, and the sender would get an exception if it tries to do so. Villard et al. [26] and Bono et al. [1] employ static analysis techniques to determine when a message can be passed by reference rather than by value. Buntinas et al. [4], Graham and Shipman [11], as well as Brightwell [2] present techniques to allocate and use shared memory for messages.

7 Conclusion

Passive processors extend SCOOP's message-passing foundation with support for safe data sharing, reducing execution time by several orders of magnitude on data-intensive parallel programs. They are useful whenever multiple processors access shared data using frequent and short read or write operations, where the overhead outweighs the benefit of asynchrony. Passive processors can be implemented with minimal effort because much of the existing infrastructure can be reused. The feature call primitive unifies sending messages to active processors and accessing shared data on passive processors. Therefore, no significant code change is necessary to set a processor passive or active. This smooth integration differentiates passive processors from other approaches. The concept of passive computational units can also be applied to other message-passing concurrency models. For instance, messages to passive actors [14] can be translated into direct, synchronous, and mutually exclusive accesses to the actor's data.

Passive processors currently do not offer shared read locks, which allow multiple clients to simultaneously operate on a passive processor. Shared read locks require features that are guaranteed to be read-only. Functions could serve as a first approximation since they are read-only by convention. Further, passive processors are not distributed yet. Because frequent remote calls are expensive, implementing distributed passive processors requires an implicit copy mechanism to move the supplier's data into the client's memory.

References

1. Bono, V., Messa, C., Padovani, L.: Typing copyless message passing. In: European Symposium on Programming. pp. 57–76 (2011)
2. Brightwell, R.: Exploiting direct access shared memory for MPI on multi-core processors. International Journal of High Performance Computing Applications 24(1), 69–77 (2010)
3. Bull, J.M., Enright, J.P., Guo, X., Maynard, C., Reid, F.: Performance evaluation of mixed-mode OpenMP/MPI implementations. International Journal of Parallel Programming 38(5–6), 396–417 (2010)
4. Buntinas, D., Mercier, G., Gropp, W.: Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. Parallel Computing 33(9), 634–644 (2007)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Springer (2007)

6. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design. Addison-Wesley, 5th edn. (2011)
7. Dunn, I.N., Meyer, G.G.: Parallel QR factorization for hybrid message passing/shared memory operation. Journal of the Franklin Institute 338(5), 601–613 (2001)
8. Ericsson: Erlang/OTP system documentation. Tech. rep., Ericsson (2012)
9. ETH Zurich: EVE. `https://trac.inf.ethz.ch/trac/meyer/eve/` (2014)
10. ETH Zurich: SCOOP executable formal specification repository. `http://bitbucket.org/bmorandi/` (2014)
11. Graham, R.L., Shipman, G.M.: MPI support for multi-core architectures: Optimized shared memory collectives. In: European PVM/MPI Users Group Meeting. pp. 130–140 (2008)
12. Gruber, O., Boyer, F.: Ownership-based isolation for concurrent actors on multi-core machines. In: European Conference on Object-Oriented Programming. pp. 281–301 (2013)
13. Gustedt, J.: Data handover: Reconciling message passing and shared memory. In: Foundations of Global Computing (2005)
14. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: International Joint Conference on Artificial Intelligence. pp. 235–245 (1973)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
16. International Organization for Standardization: Ada. Tech. Rep. ISO/IEC 8652:2012, International Organization for Standardization (2012)
17. Jones, D.L.: Decimation-in-time (DIT) radix-2 FFT. `http://cnx.org/content/m12016/1.7/` (2014)
18. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley (2004)
19. Message Passing Interface Forum: MPI: A message-passing interface standard. Tech. rep., Message Passing Interface Forum (2012)
20. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edn. (1997)
21. Morandi, B., Schill, M., Nanz, S., Meyer, B.: Prototyping a concurrency model. In: International Conference on Application of Concurrency to System Design. pp. 177–186 (2013)
22. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
23. OpenMP Architecture Review Board: OpenMP application program interface. Tech. rep., OpenMP Architecture Review Board (2013)
24. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing. pp. 427–436 (2009)
25. Schill, M., Nanz, S., Meyer, B.: Handling parallelism in a concurrency model. In: International Conference on Multicore Software Engineering, Performance, and Tools. pp. 37–48 (2013)
26. Villard, J., Étienne Lozes, Calcagno, C.: Proving copyless message passing. In: Asian Symposium on Programming Languages and Systems. pp. 194–209 (2009)
27. Wei, F., Yilmaz, A.E.: A hybrid message passing/shared memory parallelization of the adaptive integral method for multi-core clusters. Parallel Computing 37(6), 279–301 (2011)