Light Weight Concurrency in OCaml: Continuations, Monads, Promises, Events

Christophe Deleuze

Laboratoire de Conception et d'Intégration des Systèmes,
50 rue Barthélémy de Laffemas
26902 Valence Cedex 09
France
christophe.deleuze@lcis.grenoble-inp.fr

Abstract. We explore various methods to implement (very) light weight concurrency in OCaml, in both *direct* and *indirect* style. We try in particular to examine the relations between all these various schemes. Three simple example applications allow us to examine both the coding styles and the performances. The cost of context switching, thread creation and the memory footprint of a thread are compared. All implementations are much better than system and VM threads, but the "trampoline" scheme seems to be the best both at CPU and memory requirements.

1 Introduction

Concurrency is a property of systems in which several computations are executing "simultaneously", and potentially interacting with each other. Concurrency doesn't imply that some hardware parallelism be available but just that the computations (that we'll call "threads" in this text) are "in progress" at the same time, and will evolve independently. Of course, threads also need to synchronize and exchange data.

OCaml[15] is a functional language of the ML family. It has support for both imperative and object paradigms and can be compiled to native code or bytecode executed by a virtual machine. Its standard library includes a `Thread` module that provides concurrent execution of threads using either the operating system support for threads or support from the virtual machine (this last choice being available only for bytecode executables).

Besides making potentially easier the exploitation of hardware parallelism,[1] threads allow overlapping I/O and computation (while a thread is blocked on an I/O operation, other threads may proceed) and support a concurrent programming style. Many applications can be expressed more cleanly with concurrency.

Operating systems provide concurrency by time sharing of the CPU between different processes or threads. Scheduling of such threads is preemptive, *i.e.* the system decides when to suspend a thread to allow another to run. This

[1] Which OCaml threads currently mostly can't: only one running thread can access the heap. *OCaml for MultiCore* [3] is an unofficial project to remove this limitation.

operation, called a context switch, is relatively costly[16]. Since threads can be switched at any time, synchronisation tools such as locks must generally be used to define atomic operations. Locks are low level objects with ugly properties such as breaking compositionnality [22]. The same applies for OCaml VM threads.

Threads can also be implemented in user-space without support from the operating system. Either the language runtime [1] or a library [7, 6] arranges to schedule several "threads" running in a single system thread. In such a setting, scheduling is generally cooperative: threads decide themselves when to suspend to let others execute. Care must be taken in handling I/O since if one such thread blocks waiting for an I/O operation to finish the whole set of threads is blocked. The solution is to use only non blocking I/O operations and switch to another thread if the operation fails. This approach:

- removes the need for most locks (since context switches can only occur at predefined places, race conditions can (more) easily be avoided),
- allows systems with very large numbers of potentially tightly coupled threads (since they can be very light weight both in terms of memory footprint per thread and computation time per context switch),
- can be used to give the application control on the scheduling policy [17].

On the other hand the programmer has to ensure that threads do indeed yield regularly.

In this paper we study several ways to implement very light weight cooperative threads in a functional language such as OCaml without any addition to the language. We first describe our concurrency model in Section 2. Three simple example applications making heavy use of concurrency are then presented in Section 3. The various implementations are described in Sections 4 and 5. Performance comparisons based on the example applications are given in Section 6. The paper closes with a conclusion and perspectives (Section 7).

2 Concurrency Model

We define our concurrency model with a set of primitive operations for running threads allowing communication and synchronization. We first define here informally these operations as their signatures will depend on each implementation.

2.1 Basic Operations

We define the following basic operations for running threads:

spawn takes a thunk and creates a new thread for executing it. The thread will actually start running only when the *start* operation is invoked.
yield suspends the calling thread, allowing other threads to execute.
halt terminates the calling thread. If the last thread terminates, *start* returns.
start starts all the threads created by *spawn*, and waits.

stop terminates all the threads. *start* returns, that is, control returns after the call to *start*.

Most systems providing threads do not include something like the *start* operation: threads start running as soon as they are spawned. In our model, the calling ("main") code is not one of the threads but is suspended until all the threads have completed. It is then easy to spawn a set of threads to handle a specific task and resume to sequential operations when they are done. However, this choice has little impact on most of what we say in the following.

2.2 Communication

We allow threads to communicate through MVars and synchronous Fifos. Introduced in Haskell [20], MVars are shared mutable variables that provide synchronization. They can also be thought as one-cell synchronous channels. An MVar can contain a value or be empty. A tentative writer blocks if the MVar already contains a value, otherwise it writes the value and continues. A tentative reader blocks if it is empty, otherwise it takes (removes) the value and continues. Of course blocked readers or writers should be waken up when the MVar is written to or emptied.

We define α *mvar* as the type of MVars storing a value of type α. The following operations are defined:

make_mvar creates a fresh empty MVar.
put_mvar puts a value into the MVar.
take_mvar takes the value out of the MVar.

In the following we assume that only one thread wants to write and only one wants to read in a given MVar. This is not a necessary restriction, but does simplify the implementations a bit.

Contrary to an MVar, a *synchronous Fifo* can store an unlimited amount of values. Adding a value to the Fifo is a non blocking operation while taking one (the one at the head of queue) is blocking if the queue is empty. Operations are similar to MVar's:

make_fifo creates a fresh empty Fifo.
put_fifo adds a value into the Fifo.
take_fifo takes the first value out of the Fifo.

3 Three Example Applications

We now describe three example applications that will allow us to get a feel of the programming style required by our model and each of its implementations. They will also serve to collect some performance data on the various implementations.

3.1 Kpn

We consider a problem treated by Dijkstra, and solve it by a *Kahn process network*, as described in [11]. One is requested to generate the first n elements of the sequence of integers of the form $2^a 3^b 5^c$ $(a, b, c \geq 0)$ in increasing order, without omission or repetition. The idea of the solution is to think of that sequence as a single object and to notice that if we multiply it by 2, 3 or 5, we obtain subsequences. The solution sequence is the least sequence containing 1 and satisfying that property and can be computed as illustrated on Figure 1. The thread *merge* assumes two increasing sequences of integers as input and merges them, eliminating duplications on the fly. The thread *times* multiplies all elements of its input Fifo by the scalar a. Finally the thread x prints the flow of numbers and put them in the three Fifos.[2]

All threads communicate and synchronize through MVars, except that the x thread itself writes its data in three Fifos for the *times* threads to take it. The computation is initiated by *put*ting the value 1 in the *m235* MVar so that x starts running. Such a computation can be expressed as a *list comprehension* in some languages, such as Haskell[3] [21].

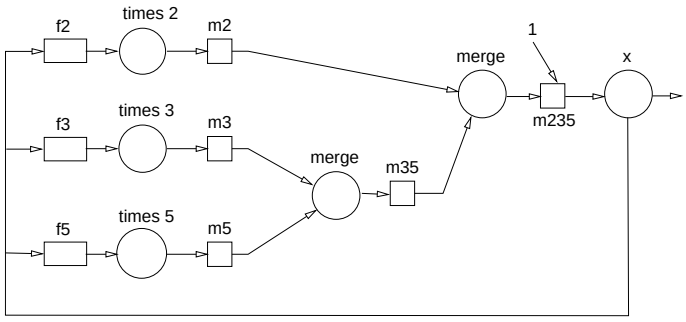


Fig. 1. Kpn (threads are circles, MVars squares, Fifos rectangles)

3.2 Eratosthene Sieve

Our second example is Eratosthene sieve. The sieve as a set of concurrent threads is described in [11] (where it is said to appear the very first time in [19]). A variant can also be found in [10]. The program is structured as a chain of threads exchanging messages.

integers is the generator, it sends out all integers starting from 2,

[2] This description is borrowed from the cited article.

[3] The Haskell code could be `s = 1:merge [ x*2 | x <- s ] (merge [ x*3 | x <- s ] [ x*5 | x <- s ])` with `merge` defined appropriately.

filter n forwards the numbers it receives that are not multiple of its n parameter,
sift creates and inserts a new filter in the chain, for each number received,
output prints the numbers it receives.

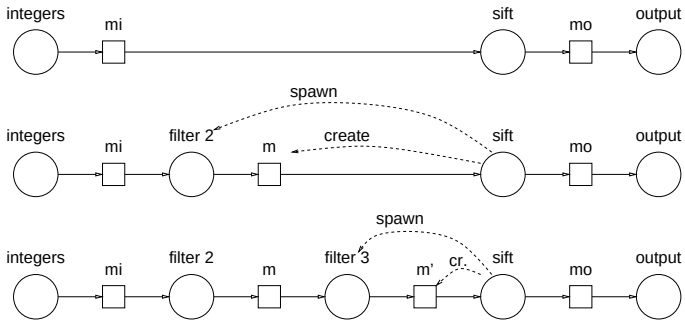


Fig. 2. Sieve as a chain of concurrent threads

Thus the sieve builds as a chain of *filter* threads with a generator (*integers*) on the left and an expander (*sift*) preceding the consumer (*output*) on the right. Threads communicate through MVars. Figure 2 shows the threads at startup and after the first and second prime numbers have been found.

The code, shown in Figure 19 in both direct and indirect style (we'll explain the distinction when presenting the implementations), is particularly simple.

3.3 Concurrent Sort

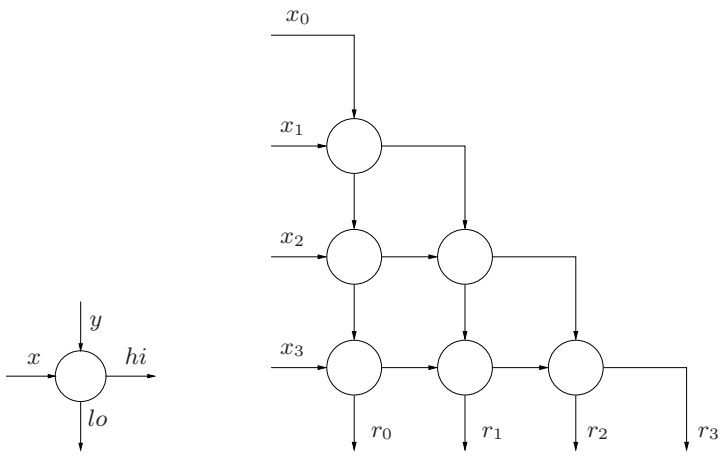
Our third example is a concurrent sort described in [10]. As pointed out by the authors, both bubble sort and insertion sort are sequentialized versions of this concurrent algorithm.

This sorting algorithm is made of a network of simple *comparator* threads, each of which is used to sort a pair of values from two input MVars to two output MVars. Such a comparator with inputs x and y and outputs hi and lo is shown on Figure 3(a). Figure 3(b) shows a network for sorting a group of 4 values.

MVars will be used to store the initial and final values, as well as for the communication between the comparators. They are not shown on the figure. We don't show the source code of the concurrent sort to save space.

4 Direct Style Implementations

When programming in *direct style* the primitives will have the signatures shown on Figure 4. Note that some of the operations are potentially blocking. The source code for the sieve is shown on Figure 19(a).



(a) Comparator (b) Sorting network

Fig. 3. Concurrent sort

In direct style, a potentially blocking operation looks just like an ordinary function. This is what is provided by preemptive scheduling systems since preemption can occur anywhere. Thus, standard OCaml support for threads can be used to implement our concurrency model in direct style. We don't describe this implementation since its only purpose is to be able to measure how light our light weight implementations really are.

```
val yield : unit → unit
val spawn : (unit → unit) → unit
val halt : unit → unit

val start : unit → unit
val stop : unit → unit
```

```
type α mvar
val make_mvar : unit → α mvar
val take_mvar : α mvar → α
val put_mvar : α mvar → α → unit
```

Fig. 4. Direct style signatures for the thread primitives

4.1 Capturing Continuations

We want to be able to suspend a running thread and activate it again later, so we need some way to save the current thread state, or rather *continuation*. The continuation of a computation at some point is what remains to be done at this point, in other words the rest of the computation. It is represented by the *context* of the computation [4]. The control flow of a program can be treated in terms of continuations.

The *call with current continuation* primitive (often abreviated as *call/cc*) was first defined in Scheme [12]. It captures (makes a copy of) the current continuation and *reifies* (makes it available in the program) it into a value of type α *cont*. Thus, continuations, which in most languages are implicit, can be explicitly captured and manipulated (passed as parameters, saved in data structures etc) like any other value. These "first class continuations" can also be *thrown*[4](and given a parameter of type α), meaning that the current continuation is discarded and replaced with the thrown one, so that execution resumes at the point where the continuation was captured. Continuations can be used to implement all sorts of manipulations of the control flow, including multi-threading.

This implementation is designed along the lines described in [8]. The queue (of type *queue_t*) stores the continuations of the suspended threads but also the initial continuation *e* (of the call to *start*) to be thrown when the queue becomes empty so that control returns after the call to *start*.

```
type α t = α → unit
type queue_t = { mutable e :unit t; q :unit t Queue.t }

let q = { e = (fun() → ()); q = Queue.create () }

let enqueue t = Queue.push t q.q
let dequeue () = try Queue.take q.q with Queue.Empty → q.e
```

There is no scheduler proper, rather each yielding thread captures its continuation with *callcc*, packages and enqueues it before dequeueing and throwing the next one. *spawn* inserts *halt* at the end of a new thread, so that it dequeues and throws the next thread continuation when it terminates.

```
exception Stop
let stop () = raise Stop

let start () =
  try
    callcc (fun exitk →
      q.e ← (fun () → throw exitk ());
      dequeue () ())
  with Stop → ()

let yield () =
  callcc (fun k → enqueue (fun () → throw k ()); dequeue () ())

let halt () = dequeue () ()

let spawn p = enqueue (fun () → p (); halt ())
```

We implement MVars as a struct containing three option values (they may be empty): the value stored in the MVar, the continuation of the thread blocked on a *take_mvar* operation, the continuation of the thread blocked on a *put_mvar* operation along with the value it wanted to put. Suppose a thread blocks on

[4] Our description applies to a statically typed language such as OCaml. In Scheme captured continuations are actually reified into functions, the continuation is *thrown* by applying the function.

put_mvar, its continuation is captured by *callcc*, packaged and stored in the MVar *write* field. The call to *halt* does not halt anything but ensures the next thread is resumed. When a thread later performs *take_mvar* on the same MVar it removes the packaged continuation from the *write* field and enqueues it to be run. The code for Fifos is obviously similar.

```
type α mvar = { mutable v : α option;
                mutable read : α t option;
                mutable write : (unit t × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
  match out with
  | { v = Some v; read = _; write = None } →
      callcc (fun k →
        out.write ← Some ((fun () → throw k ()), v); halt ())

  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → r v)

  | { v = None; read = None; write = None } → out.v ← Some v; ()

  | _ → failwith "failed put_mvar"

let take_mvar inp =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; v

  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; v

  | { v = None; read = None; write = _ } →
      callcc (fun k →
        inp.read ← Some (fun v → throw k v);
        Obj.magic halt ())

  | _ → failwith "failed take_mvar"
```

Note that we have to fool the typechecker with *Obj.magic* in *take_mvar* (and *take_fifo*) to ensure these functions are polymorphic. Otherwise, the call to *halt* makes it decide the function must return *unit* and the MVars loose their polymorphism.

4.2 Delimited Continuations

callcc captures a whole continuation. An alternative is to capture *delimited continuations*, as provided by the `caml-shift` library [23]. A delimited continuation (also called partial, composable, or sub continuation), is a prefix of the rest of the computation, represented by a delimited part of the context of the computation. Unlike regular continuations, delimited continuations return a value, and thus may be reused and composed.

Several slightly different operators have been proposed in the litterature but the general idea is that such a continuation is delimited by first pushing a delimiter (often called a *prompt*) on the stack, and later capturing the continuation, up to the first prompt.

In this library, *push_prompt* pushes a prompt on the stack, marking the delimitation, while *take_subcont* turns the part of the stack up to (and not including) the first prompt into a (α, β) *subcont* value and removes it (along with the prompt) from the stack.[5] Here α is the type of values that must be given when throwing the continuation, and β is the type of values returned by the continuation. *push_subcont* pushes a delimited continuation on the stack (*i.e.* throws it).

We can now use a simple FIFO queue (as provided by the OCaml `Queue`) to implement our scheduler:

```
let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

exception Stop
let stop () = raise Stop
```

```
let start () =
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()
```

```
let prompt = new_prompt ()

let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt_subcont p sk (fun () → c))))

let yield () = shift0 prompt (fun f → enqueue f)

let halt () = shift0 prompt (fun f → ())
```

Basically, when suspending a thread we need to capture the continuation and store it. When resuming a thread we need to re-push the prompt and the subcont. When capturing a continuation we will package it in a function that pushes the prompt and this continuation. This is the behavior of the *shift0* operator [4]. The scheduler just needs to run the packaged function for resuming a thread (thus the *dequeue* () () in the *start* loop above).

halt removes any trailing context along with the prompt to "clean up" the stack. The *shift0* definition above is an optimized[6] version of

```
let shift0 p f = take_subcont p (fun sk () →
  (f (fun c → push_prompt p
       (fun () → push_subcont sk (fun () → c)))))
```

The *spawn* function (that adds a thread in the queue) packages its argument thunk so that it first *push_prompt* and calls *halt* at the end to ensure the prompt is removed when the thread terminates.

[5] This is the behavior of the operator know as *control0* [13].

[6] Actually, the unoptimized version has a subtle memory leak (see [14, Appendix B]).

```
let spawn t = enqueue (fun () → push_prompt prompt (fun () → t (); halt ()))
```

The code for MVars is similar to the one using *callcc*. The only difference, beside the continuation being captured by *shift0*, is that the thread simply returns to the scheduler that will itself resume the next thread.

```
type α t = α → unit
type α mvar = { mutable v : α option;
                mutable read : α t option;
                mutable write : (unit t × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      shift0 prompt (fun f → out.write ← Some (f, v))
  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → r v)
  | { v = None; read = None; write = None } → out.v ← Some v

let take_mvar inp =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; v
  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; v
  | { v = None; read = None; write = _ } →
      shift0 prompt (fun f → inp.read ← Some f)
```

5 Indirect Style Implementations

The basic idea of *indirect style* is to write the thread code so that the continuations are made explicit (as closures) at each potentially blocking point. This way, the continuation can be manipulated without the need for any continuation-capture primitive.

Look at the code of the *sift* thread of the sieve on Figure 19(b). We use >>= (pronounced *bind*) as the thread sequential composition operator.[7] This operator appears at cooperation points, between a potentially blocking operation and its continuation. The continuation is a closure that will be executed when the blocking operation will have completed. The parameter of the continuation will receive the result of the operation. Imperative loops must be turned into (tail-) recursive functions if they contain a blocking operation. One more operation is useful in indirect style : *skip*, the no-op.

Trampolined style (derived from *continuation passing style*), monadic style based either on continuations or on *promises* and event-based programming are all variants of the indirect style. We describe implementations of our concurrency model in all of them in the following.

[7] This is borrowed from monad syntax but does not necessarily represent here "the" bind operator from monads, as we'll see.

5.1 Trampolined Style

The idea of trampolined style [9] is that the code is written so that a potentially blocking function is given explicitly its continuation as a closure. It can then manipulate it just like the direct style continuation-capture based versions. The code must be written in a way similar to continuation passing style [25], but the continuations need to be made explicit only at cooperation points.

Figure 5 shows the signatures of the operations. As we can see, each potentially blocking operation is given (as an additional parameter) the (continuation) function to run when the operation has been performed.

```
val skip : (unit → unit) → unit
val ( >>= ) : ((α → unit) → unit) → (α → unit) → unit

val yield : (unit → unit) → unit
val spawn : (unit → unit) → unit
val halt : unit → unit

val start : unit → unit
val stop : unit → unit

type α mvar
val make_mvar : unit → α mvar
val take_mvar : α mvar → (α → unit) → unit
val put_mvar : α mvar → α → (unit → unit) → unit
```

Fig. 5. Trampolined style signatures for the thread primitives

For example the *yield* operation can be used as shown below on the left where the argument is the continuation, *i.e.* the function to be executed when the thread will be resumed. By defining "bind" (infix >>=) to take two arguments and apply its first to the second (the continuation) we obtain a arguably more pleasant syntax.[8] Adopting an indentation more fitted to the intended "sequential execution" semantics, the code can now be written as below on the right.

```
print_string "hoho";
  yield (fun () →
    print_string "haha";
    yield (fun () →
      ...
        ))
```

```
let (>>=) inst (k : α → unit) : unit = inst k
  ...
  print_string "hoho";
  yield >>= fun () →
  print_string "haha";
  yield >>= fun () →
  ...
```

Writing code so that continuations are explicit is often not as intrusive as one may feel initially. As the code for the sieve shows, continuations often do not even appear explicitly. Actually, only when using procedural abstractions to build complex blocking operations do we need to manipulate explicitly the extra parameter. Even then it is easy, as the following two functions show. The first one abstracts the operation of *yield*ing three times and the second one the transfer of a value from an MVar to another one:

[8] Haskell programmers will think of the $ operator.

```
let yield3 k =
  yield >>= fun () →
  yield >>= fun () →
  yield >>=
  k
```

```
let transfer_mvar m1 m2 k =
  take_mvar m1 >>= fun v →
  put_mvar m2 v >>=
  k
```

Implementation Since a potentially blocking function, such as *yield* or *take_mvar*, takes its continuation as an additional parameter, it can execute it immediately or, if it needs to block, store it for later resuming before returning to the scheduler.

The code is very similar to the one using delimited continuations: the only difference is we don't have to capture continuations as they are provided explicitly as illustrated below:

```
let skip k = k ()
let yield k = enqueue k
let halt () = ()
```

```
let spawn t = enqueue t
let close k = fun () → k (fun _ → ())
```

```
let take_mvar inp k =
  match inp with
  | { v = Some v; read = None; write = None } → inp.v ← None; k v
  | { v = Some v; read = None; write = Some(c, v') } →
      inp.v ← Some v'; inp.write ← None; enqueue c; k v
  | { v = None; read = None; write = _ } → inp.read ← Some(k)
```

A small note about *spawn*. We can see that composing fragments with >>= produces a function accepting a continuation that needs, before to be executed, to be given a final dummy continuation. *spawn* could do that, but as the sieve example shows, when composing recursively we may omit the thread continuation argument. We thus define *spawn* as acting on thunks, and define a *close* function building a thunk by providing the dummy continuation.

5.2 Continuation Monad

Trampolined style has been created in the dynamically language Scheme. As we'll see, the continuation monad [5] is basically a formulation of the same ideas in a statically typed language. But let's first quicky define monads.

Monads are useful in a variety of situations for dealing with effects in a functional setting [20]. A monad is a (parameterized) data type $\alpha\ t$ with (at least) the *return* and *bind* (noted infix >>=) primitives whose types are shown on Figure 6. As the types suggest, *return* v builds a monadic $\alpha\ t$ value "containing" v and m >>= f "opens up" m to get its enclosed value v, give it to f and return the monadic value returned by f.

Here we can define type $\alpha\ t = (\alpha \to unit) \to unit$ so that a "monadic value" $\alpha\ t$ is a function taking an $\alpha \to unit$ continuation. We can now think of blocking functions as returning a value of type $\alpha\ t$ as can be seen on Figure 6.

```
type α t
val return : α → α t
val ( >>= ) : α t → (α → β t) → β t

val spawn : (unit → unit t) → unit
val skip : unit t
val yield : unit → unit t
val halt : unit → unit t
val stop : unit → unit t
val start : unit → unit

type α mvar
val make_mvar : unit → α mvar
val put_mvar : α mvar → α → unit t
val take_mvar : α mvar → α t
```

Fig. 6. Monadic style signatures for the thread primitives

This *bind* is obviously different from the trivial application operator that we used in trampolined style. Apart from ensuring there's one argument to all functions (besides the continuation) it removes the need to explicitly manage the continuation parameter when composing thread fragments. Here's our *yield3* example (we have cosmetically changed the definition of *yield* so that it takes a () argument to be compatible with the new *bind*):

```
let yield3 () =
  yield () >>= fun () →
  yield () >>= fun () →
  yield ()
```

```
let transfer_mvar m1 m2 =
  take_mvar m1 >>= fun v →
  put_mvar m2 v
```

Just to be explicit about how this works, the "expanded" types and definitions of *return* and *bind* are:

```
val return : α → (α → unit) → unit
let return a = fun k → k a

val (>>=) : ((α → unit) → unit) → (α → (β → unit) → unit) → (β →
 unit) → unit
let (>>=) f k′ = fun k → f (fun r → k′ r k)
```

that is, *bind* returns a function accepting a continuation. Let's see this on our *transfer_mvar* example by expanding its definition:

```
transfer_mvar m1 m2 ≜
      fun k → take_mvar m1 (fun r → (fun v → put_mvar m2 v) r k)
```

Finally, *spawn* provides the final dummy continuation to get a thunk it then enqueues.

```
let spawn (t : unit → unit t) = enqueue (fun () → t () (fun () → ()))
```

5.3 Promise Monad

A *promise* [18] is a "proxy" value that can be used to later access a value that is not immediately available. Using promises, blocking operations need not block: they return immediately a promise for the requested value. A promise can either

be *ready* if the value is indeed available or *blocked* if it's still not. The promise can be passed along until its value is actually needed: the *claim* operation will then be used to get the promised value. Of course *claim* itself may block.

Using the promise monad to implement threading, operations will still have the signatures of Figure 6. Blocking operations will return a promise and the duty of *claim* will be performed by *bind*. In t >>= f, *bind* will either:

- pass t's value to f if it's ready (thus returning the promise returned by f),
- or return a blocked promise and set up things so that:
  - f is passed the promised value as soon as t becomes ready,
  - the returned blocked promise is "the one" returned by f.

Implementation Here's a brief description of how it can be implemented (Figure 7). This implementation is heavily inspired by `Lwt` (light weigth threads) [26], a cooperative threads library for OCaml. Our implementation tries to retain only the core workings of Lwt so that it can be compared to the other implementations. Also, [26] describes it differently, by interpreting α t as the type of threads rather than that of promises.

α t is the type of promises for a value of type α. It is a record with one mutable field denoting the current promise state. It can be *Ready* and containing the promised value, *Blocked* and holding a list of thunks (the "waiters") to execute when it will be ready, or *Link*ed to another one (that means it will behave the same, having been *connect*ed to it). *repr* gets the promise a given promise is *Link*ed to.

Consider the evaluation of t >>= f, following the code for >>=. Evaluating t provides a promise. If it's ready its value v is passed to f. If it's blocked, its value is *Blocked* w. A new blocked promise *res* is created, a thunk is added to the w list, then *res* is returned.

Thus, when t becomes ready (through *fullfill*), the thunk is executed. It passes the value to f which returns a new promise (let's call it p). *res* is then *connect*ed to p. The code for *connect* shows that if p is ready *res* is *fullfill*ed. If p is blocked it is changed as a *Link* to *res*: this ensures any later operation (such as *fullfill*) involving p will be actually performed on *res*.

Figure 8 shows that the scheduler first *fullfill*s the *wait_start* promise. *spawn* makes threads wait on it to ensure they don't start running before allowed to. The code for *yield* shows how we suspend a thread: we create a blocked promise p, enqueue a function that will fullfill it and return p. *bind* will add to p a thunk *connect*ing the promise of the next computation to its own *res*. When the scheduler dequeues the function enqueued by *yield*, it fullfills p and so executes the thunk. The same applies for MVars.

It's not obvious to see how operations chain, so we illustrate the execution of our *yield3* example on Figure 9. A promise is shown as a square (R for ready, B for blocked, pointing to its waiter thunk if any, L for a link). (a) In the first occurence of *bind* (subscripted 1 for convenience), the first *yield* (subscripted a) returns a blocked promise p_a. *bind* adds it a waiter thunk and returns a fresh blocked promise res_1. Remember *yield* has enqueued a thunk to fullfill p_a.

```
type α state =
  | Ready of α
  | Blocked of (α t → unit) list ref
  | Link of α t

and α t = { mutable st : α state }

let rec repr t =
  match t.st with
  | Link t' → repr t'
  | _ → t

let blocked () = { st = Blocked (ref []) }
let ready v = { st = Ready v }

let runq = Queue.create ()
let enqueue t = Queue.push t runq
let dequeue () = Queue.take runq

let fullfill t v =
  let t = repr t in
  match t.st with
  | Blocked w →
      t.st ← Ready v;
      List.iter (fun f → f t) !w
  | _ → failwith "fullfill"

let connect t t' =
  let t' = repr t' in
  match t'.st with
  | Ready v → fullfill t v
  | Blocked w' →
      let t = repr t in
      match t.st with
      | Blocked w → w := !w @ !w';
                    t'.st ← Link t
      | _ → failwith "connect"

let (>>=) t f =
  match (repr t).st with
  | Ready v → f v
  | Blocked w → let res = blocked () in
      w := (fun t → let Ready v = t.st in
                    connect res (f v)) :: !w;
      res
```

Fig. 7. Promise monad: core

(b) When this occurs, p_a's waiter thunk is executed. $yield_b$ returns a blocked promise p_b with the rest of the computation as waiter thunk. $bind_2$ returns a new blocked promise res_2 which is *connect*ed to res_1. Since res_2 is blocked, it is turned into a link to res_1. (c) When p_b's waiter thunk is executed, $yield_c$ returns still a blocked promise p_c, that is turned into a link to res_1 (since res_2 is itself a link to res_1) by *connect*. (d) Finally, when p_c is fullfilled, res_1 becomes ready.

5.4 Event-based Programming

A popular paradigm supporting user level concurrency is event-driven programming. The OCamlNet library [24] provides an `equeue` (for *event queue*) module in which handlers (or *callbacks*) are set up to process events.

We describe it briefly. First an *event system* (called *esys* here) must be created. Events are generated by an event source (here it is the function `fun _ → ()` that generates none) but can also be added by the handlers themselves. Each event in presented to each handler, in turn, until one accepts it (or it is dropped if no one does). An handler rejects an event by raising the *Reject* exception. Otherwise the event is accepted. In case the handler, having accepted the event, wants to remove itself it must raise the *Terminate* exception.

The event system is activated by the *Equeue.run* function. The function returns when all events have been consumed and the event source does not add any.

```
let skip = ready ()
let halt () = ready ()
let yield () = let p = blocked () in enqueue (fun () → fullfill p ()); p
let wait_start = blocked ()
let spawn t = wait_start >>= t; ()

exception Stop
let stop () = raise Stop

let start () =
  fullfill wait_start ();
  try
    while true do
      dequeue () ()
    done
  with Queue.Empty | Stop → ()

type α mvar = { mutable v : α option;
                mutable read : α t option;
                mutable write : (unit t × α) option }

let put_mvar out v =
  match out with
  | { v = Some v'; read = _; write = None } →
      let w = blocked () in out.write ← Some (w, v); w

  | { v = None; read = Some r; write = None } →
      out.read ← None; enqueue (fun () → fullfill r v); ready ()

  | { v = None; read = None; write = None } → out.v ← Some v; ready ()
```

Fig. 8. Promise monad: scheduler, MVars

In our implementation, handlers will always be "one shot", so they will always raise *Terminate* after having accepted an event. But before to do that, they will have registered a new handler representing the thread continuation.

For *yield*ing, a thread creates a new *eventid*, registers its continuation as a handler to the *Go* event with the correct id, and adds this precise event to the system.

```
let skip k = k ()

let (>>=) inst k = inst k

type eventid = unit ref
type α event = Written of eventid
| Read of eventid × α | Go of eventid

let make_eventid () = ref ()

let esys : int event Equeue.t =
  Equeue.create (fun _ → ())
```

```
let yield k =
  let id = make_eventid () in
  Equeue.add_handler esys
    (fun esys e →
      match e with
      | Go id' when id' ≡ id → k ()
      | _ → raise Equeue.Reject);
  Equeue.add_event esys (Go id);
  raise Equeue.Terminate
```

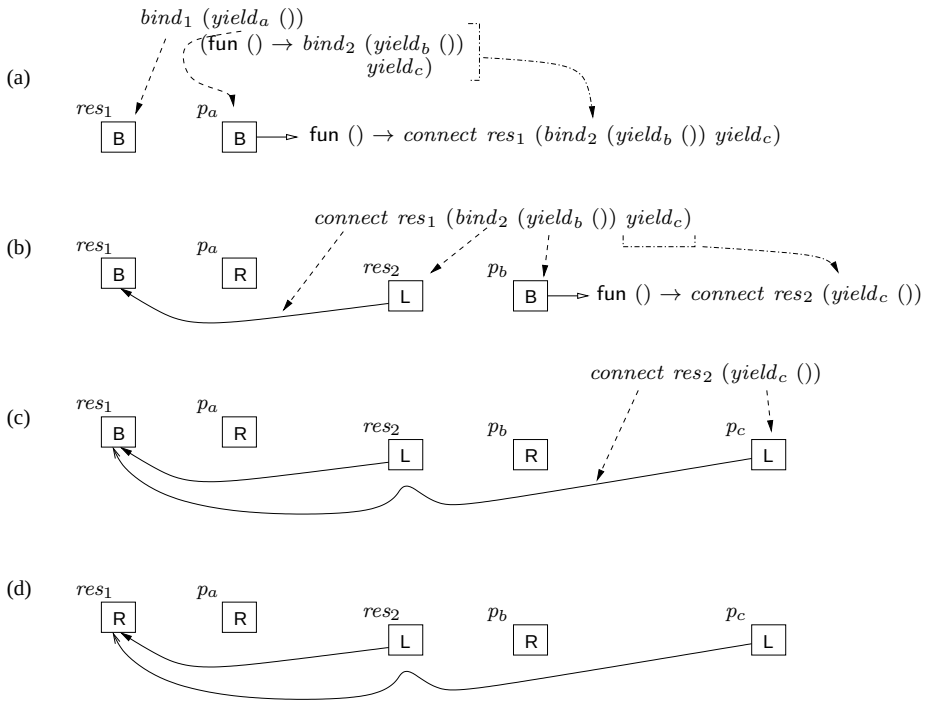


Fig. 9. Execution of *yield3*

A thread blocked on a MVar waits for a unique event allowing it to proceed. Blocked writers create a new *eventid* that they register in the control information of the MVar, along with the value they want to write. They then wait for a *Written* event with the correct id. Such an event will be generated when the value will have been actually put in the MVar, operation triggered by the *take*ing of the current MVar value by another thread. Blocked readers create a new *eventid* and wait for the *Read* event that will carry the value taken from the MVar. Again, this event will be generated when some thread puts a value in the MVar.

```
type α mvar = { mutable v : α option;
                mutable read : eventid option;
                mutable write : (eventid × α) option }

let make_mvar () = { v = None; read = None; write = None }

let put_mvar out v k =
  match out with
  | { v = Some v'; read = _; write = None } →
      let id = make_eventid () in out.write ← Some (id, v);
      Equeue.add_handler esys (fun esys e →
        match e with
        | Written id' when id' ≡ id → k ()
        | _ → raise Equeue.Reject);
```

```
      raise Equeue.Terminate
  | { v = None; read = Some id; write = None } →
      out.read ← None;
      Equeue.add_event esys (Read(id, v));
      k ()
  | { v = None; read = None; write = None } → out.v ← Some v; k ()
```

Since each blocking operation (in case it actually blocks) registers a new handler and then raises *Terminate*, threads must be running as handlers from the very beginning (for the *Terminate* exception to be catched by the event system). To ensure this, *spawn* registers the new thread as a handler for a new *Go* event, then adds the event to the system.

```
let spawn t =
  let id = make_eventid () in
  Equeue.add_handler esys (fun esys e →
    match e with
    | Go id' when id' ≡ id → t ()
    | _ → raise Equeue.Reject);
  Equeue.add_event esys (Go id)
```

There's one serious pitfall with this implementation: MVar operations are not polymorphic due to the event system being a monomorphic queue. Thus, all MVars are required to store the same type of value, which is a serious limitation.

The code for the applications is strictly the same as for the trampolined implementation. Indeed, the threads are written in trampolined style and the event framework is only used to build the scheduler. This implementation can be seen more as an exercise in style.[9]

6 Performance

We have measured the time and memory needs of these implementations on the three examples. The execution times were collected by the *Unix.times* function, while the memory usage was measured as the *top_heap_words* given by the *quick_stat* function of the OCaml `Gc` module (that provides an interface to the garbage collector).

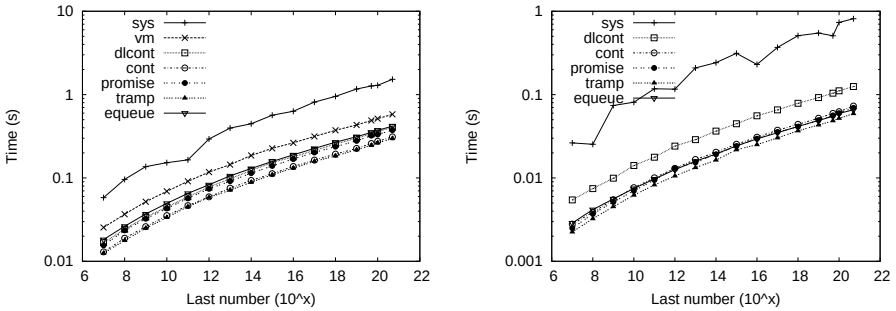
All the programs were run on a PC powered by an Intel Core 2 Duo CPU clocked at 2.33 GHz with 2 GB of memory, running the linux kernel version 2.6.26 with amd64 architecture. Software versions were OCaml 3.11.2, `caml-shift` august 2010, `equeue` 2.2.9.

All our examples use very simple threads that cooperate heavily. `kpn` has a fixed number of threads, `sieve` is interesting because it constantly creates new threads. `sorter` has both a number of threads (all initially spawn) and a number of operations depending on the problem size. Moreover, its number of threads can easily be made huge (for sorting a list of 3000 numbers, there're about 4.5 million threads). To measure thread creation time alone, we will also run it with

[9] But one could argue that **all** our implementations are!

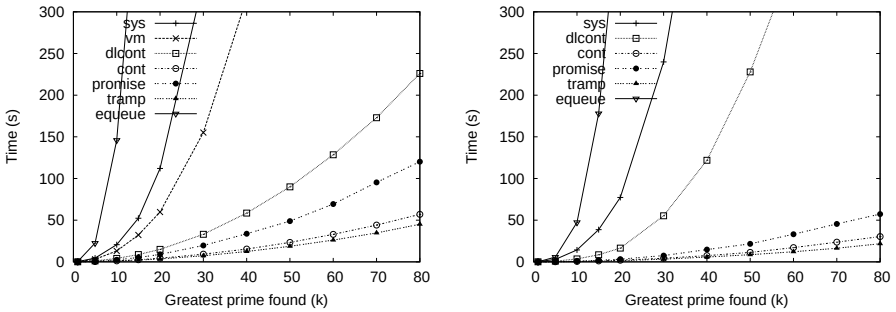
a parameter `-d` that terminates the program as soon as the sorting network has been set up.

In the following text and graphs, we'll call `sys` the implementation using system threads, `vm` the VM threads, `callcc` and `dlcont` the continuation captures, `tramp` the trampolined style, `cont` the continuation monad, `promise` the promise monad, and `equeue` the event-based one.



(a) Bytecode version (b) Native-code version

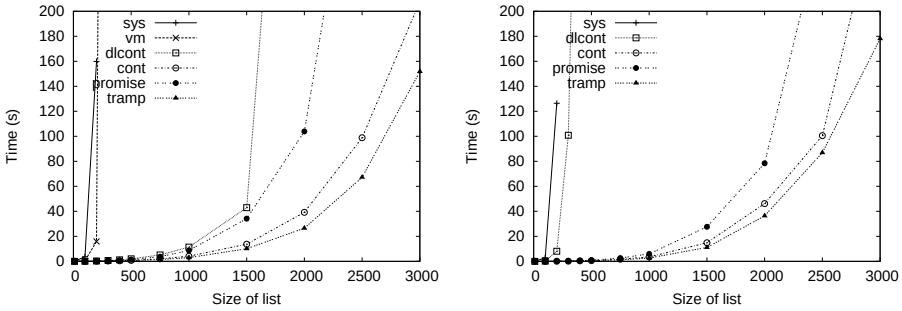
Fig. 10. kpn, execution time



(a) Bytecode version (b) Native-code version

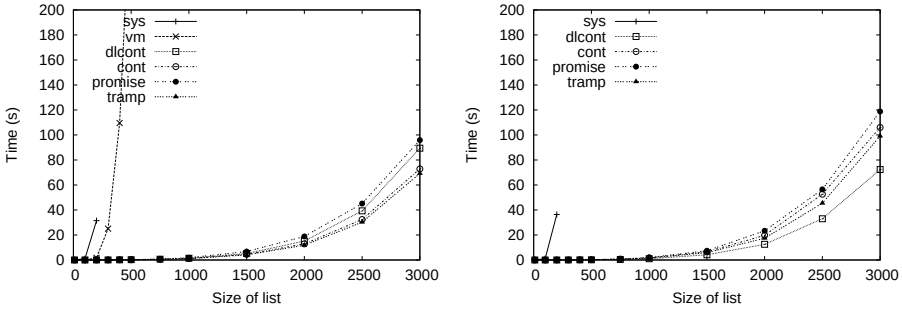
Fig. 11. sieve, execution time

Execution time Figure 10 shows the execution time for `kpn`, on a log-log graph. `callcc` is notably slow. Even heavy-weight `sys` is much faster, `vm` being ten times faster. The other implementations are rather similar, `tramp` being slightly better in the bytecode version.



(a) Bytecode version (b) Native-code version

Fig. 12. sorter, execution time



(a) Bytecode version (b) Native-code version

Fig. 13. sorter -d, setup time only

We note that `dlcont` performance is on par with `promises` and `equeue` for bytecode but slower in native code. VM threads are much better than system threads and are only slightly slower than the light weigth implementations.

Figure 11 shows the execution times for the sieve. `equeue` performance is terrible (much worse than `sys`). The problem is in the implementation of the `Equeue` module. As we said, events are presented to each handler in turn until one accepts it. In effect the threads are performing active wait on the MVars. Thus, `equeue` does not scale with the number of threads.

As a side note, it seems that a simple change in `Equeue` implementation would dramatically improve performance for the sieve: events should be presented only to (or starting with) handlers *set up after the handler that generated them*, in the order they were set up. This way each event would be presented immediately to the handler that is waiting for it. This would take advantage of the very particular communication pattern of the concurrent sieve and is not generally applicable, of course.

`vm` and `sys` are both much slower than the other light weight implementations, among which `tramp` is the fastest (with `cont` only slightly above). Time

doubles for `promise` and doubles again for `callcc` and `dlcont` in bytecode. `dlcont` performs very poorly in native code.

For sorter (Figure 12), `callcc` makes the system trash, `equeue` is not shown, while `sys` is shown only for lists of size 100 and 200. The number of threads used by sorter is about $n \times (n-1)/2$ with n the list size, which means 19900 threads for n=200 and 44850 for 300. `vm` performs very badly. Here again `tramp` and `cont` are notably better. `promises` is always significantly slower than `tramp` and `cont`. This is certainly due to the much more complex implementation of the *bind* operator.

Figure 13 shows the time to set up the sorter network but not running it. The main difference is `dlcont` and `callcc` being rather good this time. Indeed, since the threads are not running, no continuation captures are performed!

This is the only figure where the some performances for bytecode are (slighlty) better than those for native code. According to OCaml's documentation native code executables are faster but also bigger which can translate into larger startup time but this wouldn't alone explain what we see here. Memory allocation may be slower since sorter `-d` essentially allocates (a large number of) threads and MVars.

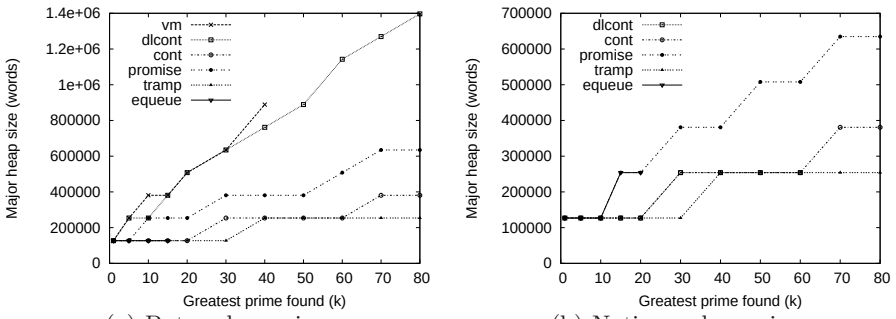
Memory usage We don't show the graphs for kpn: they are as expected with all implementations having the same constant memory requirement with the notable exception of `callcc` whose memory consumption increases roughly linearly with time. There's clearly a memory leak, but the authors had warned us of its experimental status.

The graphs for sieve are shown in Figure 14. `tramp`/`cont` are clearly the best. `equeue` is good too but values are shown only for the first few points since the program is so slow... We don't show `sys` since we should also measure the memory used by the operating system itself in its managing of threads. `vm` is hidden behind `dlcont`.

`dlcont` is much above the other implementations for sorter on Figure 15, but is quite good (as `callcc`) with sorter `-d` since again no continuation captures occur. We don't include the graphs for native code, they mainly show `promise` is better in native code than byte code.

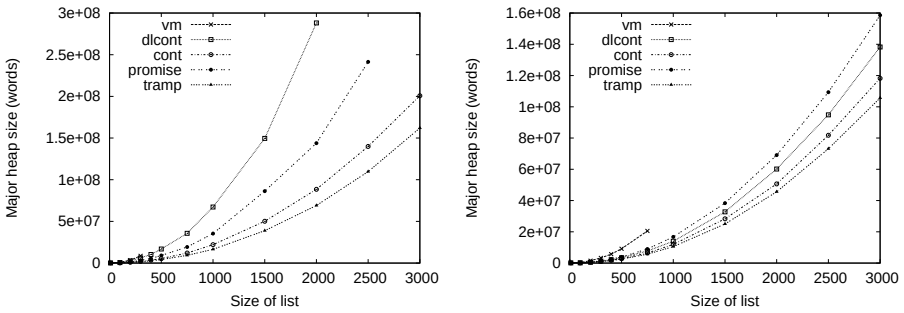
Finally Figures 16, 17, and 18 present the same data with a different view: they show the memory requirements per thread. Apart from `dlcont` that is very good in native code, `tramp` and `cont` are always under the other implementations. It's interesting to see on Figures 17 and 18 that its advantage is much larger when the threads are running. The advantage over `promise` is probably caused by the relative complexity introduced by the handling of promises.

Other comments It has been noted [2] that continuations used for implementing threads are *one shot continuations*. Specific optimized implementations may be designed for them but no such implementation currently exists for OCaml. Likewise, the `equeue` module is clearly designed with long standing handlers in mind. Since we use one shot handlers, we pay the cost of throwing an exception (to remove the current handler) at each cooperation point.



(a) Bytecode version (b) Native-code version

Fig. 14. sieve, memory usage



(a) sorter (b) sorted -d

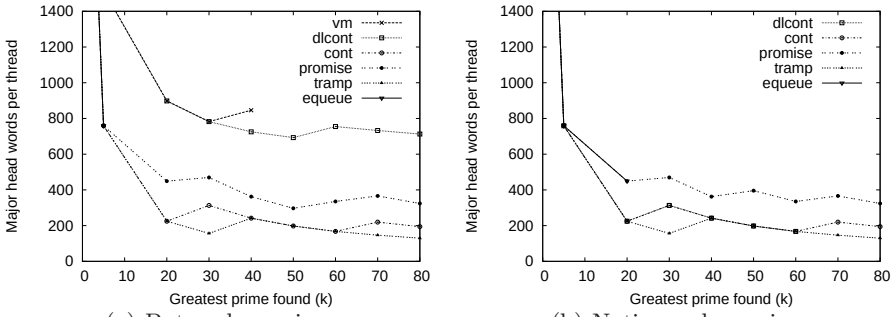
Fig. 15. sorter, memory usage for bytecode

7 Conclusion and Perspectives

We have described, implemented, and evaluated several ways to implement light weight concurrency in OCaml. The complete code for the implementations is available on the web (`http://christophe.deleuze.free.fr/lwc/`). A direct style implementation involves capturing continuations, which is relatively costly (although much less than what is incurred by VM or system threads). Indirect style implementations perform better but force the programmer to write in a specific style.

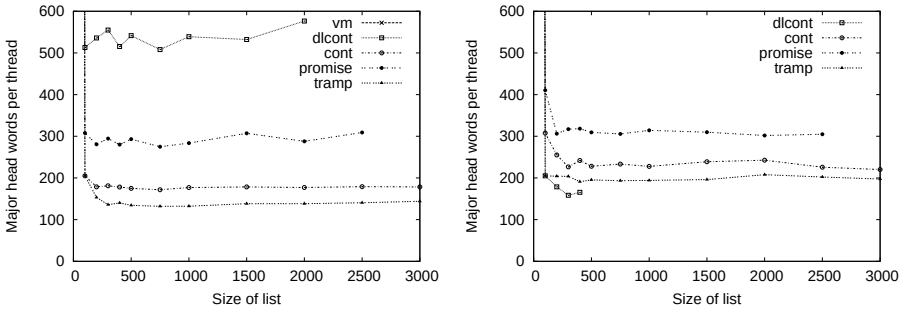
As we saw, event-based programming can be seen as a form of trampolined style programming with an event-based scheduling strategy. We didn't realize this immediately since event-based programming is mostly associated with imperative languages while trampolined style is with functional ones.

Apart from `equeue` that is not designed for massive concurrency, the light weight implementations can easily handle millions of threads. The trampolined implementation is the lighter, with the continuation monad slightly above. Using promises is significantly more costly: each cooperation point involves much



(a) Bytecode version (b) Native-code version

Fig. 16. sieve, memory usage per thread



(a) Bytecode version (b) Native-code version

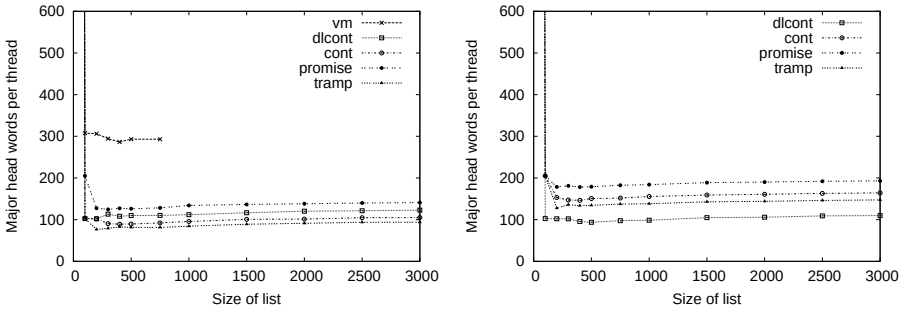
Fig. 17. sorter, memory usage per thread

more operations. Of course our examples are minimal, threads do very little work between cooperation points so any difference in the performance of their implementation is magnified. This should be kept in mind when looking at the performance results.

Our implementations are skeletons, realistic libraries should at least deal properly with I/O and exceptions. As we said `Lwt` is such a mature library based on the promise monad. We are currently developping a library (called μthreads) for light weight concurrency in OCaml, based on the trampoline scheme. More realistic applications, such as an FTP server and a DNS resolver, are also being developed.

References

1. Joe Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6.1–6.26, New York, NY, USA, 2007. ACM.
2. Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *In Proceedings of the ACM SIGPLAN'96*



(a) Bytecode version (b) Native-code version

Fig. 18. sorter -d, memory usage per thread

Conference on Programming Language Design and Implementation, pages 99–107, 1996.

3. Emmanuel Chailloux et al. OCaml4MultiCore : Objective caml for multicore architectures. web site, November 2009. `http://www.algo-prog.info/ocmc/web/`
4. Chung chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 5th workshop on Scheme and functional programming*, pages 99–107. Indiana University, 2004.
5. Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, May 1999. Functionnal Pearls.
6. Russ Cox. Libtask: a coroutine library for C and Unix. Software library. `http://swtch.com/libtask/`
7. Ralf S. Engelschall. GNU Pth - the GNU portable threads. Software library. `http://www.gnu.org/software/pth/`
8. D. P. Friedman. Applications of continuations. Invited Tutorial, Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 1988. `http://www.cs.indiana.edu/~dfried/appcont.pdf`
9. Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.
10. Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, New Haven, CT 06520-2158, 1993.
11. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information processing*, pages 993–998, Toronto, August 1977.
12. R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.
13. Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Indiana University, March 2005. `http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR611`
14. Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely system description. Technical report, March 2010. Also on FLOPS 2010. `http://okmij.org/ftp/Computation/caml-shift.pdf`
15. Xavier Leroy. *The Objective Caml system – Documentation and user's manual*. INRIA, release 3.11 edition, 2008.

```
open Lwc

let rec integers out i () =
  put_mvar out i;
  integers out (i + 1) ()

let rec output inp () =
  let v = take_mvar inp in
  if !print then Printf.printf "%i␣" v;
  if v < !last then output inp () else stop ()

let rec filter n inp out () =
  let v = take_mvar inp in
  if v mod n ≠ 0 then put_mvar out v;
  filter n inp out ()

let rec sift inp out () =
  let v = take_mvar inp in
  put_mvar out v;
  let mid = make_mvar () in
  spawn (filter v inp mid);
  sift mid out ()

let sieve () =
  let mi = make_mvar () in
  let mo = make_mvar () in
  spawn (integers mi 2);
  spawn (sift mi mo);
  spawn (output mo);
  start ()
```

(a) Direct Style

```
open Lwc

let rec integers out i () =
  put_mvar out i >>= integers out (i+1)

let rec output inp () =
  take_mvar inp >>= fun v →
  if !print then Printf.printf "%i␣" v;
  if v < !last then output inp () else (stop (); halt())

let rec filter n inp out () =
  take_mvar inp >>= fun v →
  (if v mod n ≠ 0 then put_mvar out v else skip) >>=
  filter n inp out

let rec sift inp out () =
  take_mvar inp >>= fun v →
  put_mvar out v >>= fun () →
  let mid = make_mvar () in
  spawn (filter v inp mid);
  sift mid out ()

let sieve () =
  let mi = make_mvar () in
  let mo = make_mvar () in
  spawn (integers mi 2);
  spawn (sift mi mo);
  spawn (output mo);
  start ()
```

(b) Indirect Style

Fig. 19. Implementation of the sieve

16. Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.
17. Peng Li and Steve Zdancewic. Combining events and threads for scalable network services. In *Proceedings of 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 189–199, 2007.
18. B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23:260–267, June 1988. `http://doi.acm.org/10.1145/960116.54016`
19. M. Douglas McIlroy. Coroutines. Internal report, Bell telephone laboratories, Murray Hill, New Jersey, May 1968.
20. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdof Summer School 2000*, pages 47–96. IOS Press, 2001. `http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/`

21. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
22. Simon Peyton Jones. *Beautiful code*, chapter Beautiful concurrency. O'Reilly, 2007.
23. Amr Sabry, Chung chieh Shan, and Oleg Kiselyov. Native delimited continuations in (byte-code) OCaml. OCaml library, 2008. `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`
24. Gerd Stolpmann. Ocamlnet. `http://projects.camlcity.org/projects/ocamlnet.html`
25. Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, December 1998. Reprint from AI memo 349, December 1975.
26. Jérôme Vouillon. Lwt: a cooperative thread library. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM.