An Efficient Implementation for Coroutines

Luis Mateu

INRIA-Rocquencourt &
Universidad de Chile

Abstract. Emulating coroutines with first-class continuations imposes an unacceptable overhead in managing function frames when there is an intensive exchange of control. This paper presents a high-performance implementation for a restricted class of continuations. These continuations are exploited in a simple coroutine mechanism, reaching a rate of 430,000 control exchanges per second on a modern RISC processor. As an extra feature, first-class continuations are recovered from the restricted class.
Keywords: coroutines, continuations, garbage collection, dynamic variables, shallow binding

1 Introduction

A coroutine is a kind of concurrent process, getting and passing control explicitly. The simplest way to implement them is to use multiple stacks, one for each coroutine. The problem with this approach is that whenever memory resources are limited, the deepest function recursion must be traded off against the maximal number of simultaneous coroutines. Yet, predicting the deepest function recursion is generally impossible.

On the other hand, Scheme [Rees & Clinger 86] has abstracted a wide variety of control structures —including coroutines and escapes— into just one general control operator named `call-with-current-continuation` (or, in its abbreviated form, `call/cc`). This operator reifies its *continuation* into a first-class function, which can then be treated just as any other function in Scheme. The continuation of `call/cc` is the rest of the computation from its application point. In Scheme, coroutines can be obtained by reifying the continuation of a computation to emulate the exchange of control [Haynes *et al* 86].

Scheme continuations can be implemented by allocating function frames in the general heap, where they are managed by a normal garbage collector. In this way, there is no trade off to be solved because all frames share the same heap and deep function recursion is treated by normal heap exhaustion. With some optimizations [Clinger *et al* 88], this memory organization has a small overhead for normal procedural applications.

However, we state in Sect. 2 that Scheme continuations could not be an effective way to emulate coroutines, because once a continuation has been reified, the only way to recycle the captured frames is by triggering an expensive general garbage collection, in which all the objects are involved.

* Postal address: INRIA, Bât. 8, Domaine de Voluceau-Rocquencourt, B.P. 105, 78153 LE CHESNAY CEDEX, France. Email Address: mateu@margaux.inria.fr.

The goal of this paper is to introduce a fast implementation technique for a restricted class of continuations. These continuations are used in a simple coroutine mechanism, solving effectively those problems having a natural solution with coroutines, i.e. the performances are competitive with alternative procedural solutions. If concurrency were to be added among the features of Scheme, to have a fast coroutine mechanism (i.e. context switch facility) is also of paramount importance and is solved by our model.

The basic idea is to allocate frames in a dedicated heap, managed by a generational *Stop and Copy* garbage collector. We add a new object class, the *hooks*, which are used to store the continuation of suspended coroutines. Thus continuations can be only held in hooks. When the frame memory is exhausted, an inexpensive garbage collection recycles unreachable frames. This collection is cheap since it is only applied to the frame heap compared, as in the Scheme case, to the whole general heap. This is possible, since the roots are found in the hooks, which are bounded by the number of coroutines.

In Sect. 3 we present the set of coroutine primitives and we show that they recover the Scheme notion of first class continuations. Also, we apply them to solve the *same-fringe* problem in an elegant way. In sections 4 and 5 we implement them.

In Sect. 7 we compare the performances of our heap organization to several stack organizations, showing that the main overhead comes from the locality loss in memory access. So in Sect. 8 we introduce an optimization for normal applications which reduces most of this overhead. With this optimization and others, the execution time overhead for normal applications is around 11%, compared to a stack based implementation providing no coroutine facility. In Sect. 9 we compare the performances of a coroutine based solution of the *same-fringe* problem against the trivial procedural solution. Some possible extensions are discussed in Sect. 10.

2 First-class continuations and the same-fringe problem

The *same-fringe* problem determines whether the sequence of leaves —the fringe— of two trees are the same. This problem is easily solved with three coroutines as shown in the next section. The first compares the leaf sequences returned sequentially by the other two coroutines, each of them traversing one of the trees recursively. When arriving at a leaf, a coroutine traversing a tree passes the control to the comparing coroutine. Later, the coroutine is resumed at the same point where it had been suspended, to continue traversing that tree. In this section we examine the performances of a garbage collector when first-class continuations are used to emulate suspension and resumption of the coroutines in the *same-fringe* problem.

As stated in the introduction, a trivial implementation of Scheme continuations is achieved by allocating frames in the general heap. Unfortunately, memory allocated for frames is much more important that memory allocated for normal objects so garbage collection activity increases, thus degrading performance. This heavy frame allocation is not visible in a stack organization, because frame lifetime is very short, thus frames are popped as soon as they are pushed.

For applications not using the Scheme continuations intensively, several implementation strategies are discussed in [Clinger *et al* 88] and [Hieb *et al* 90]. These

strategies reduce the associated overhead by using a stack cache to execute normal call/return behavior, but transferring frames from the stack to the general heap when a continuation is reified. In some strategies frames are also transferred from the heap to the stack when a continuation is invoked.

Now, let us consider using first-class continuations to emulate the coroutines in the *same-fringe* problem. The suspension of a tree traversal is achieved by reifying its continuation, and the resumption by invoking it. To traverse a tree recursively, a function is called at every internal node. This function allocates a frame in the stack cache. However, sooner or later that frame will be transferred to the heap by a continuation capture at a leaf. Therefore any optimization introduced to treat normal call/return behavior will be useless, because all frames will be captured by a continuation.

Considering that the size of each transferred frame is at least the size of a cons cell, and there is an additional space overhead in creating a callable continuation, we become aware that the garbage collection activity will be much more important than in a trivial solution which flattens the trees into lists prior to comparison. Thus performances will be unacceptably slow for the first-class continuation solution.

3 Coroutines as second class continuations

In fact, the aim of using coroutines to solve the *same-fringe* problem is firstly, to decrease the additional memory requirements to allocate a new list in the tree flattening solution, and secondly, to reduce the execution time overhead incurred in managing that memory. When emulating coroutines with first-class continuations, we can see that the former is successful, because the surviving frames at memory exhaustion are just those present in the branch of the current node in the tree traversal, and not the whole. Yet, for the latter, it is just the opposite that has been obtained.

Therefore, beginning with this section, we will be concerned with reducing the memory management overhead incurred to treat coroutines when frames are allocated in a heap. To achieve this goal we will introduce a coroutine definition based on continuations. Although these continuations are not first-class functions as in Scheme, we will show that `call/cc` can still be obtained from our coroutines.

We start by defining the new type *hook*. A hook is a continuation holder encapsulating a limited set of legal operations. Hooks are first-class objects, i.e. they have an unlimited extent and they can be passed as arguments to functions, returned from functions, and stored in data structures. They are created and manipulated with the following operations (an accurate semantics is presented in the appendix):

- (`coroutine f`) with `f`= (`lambda` (`hook`) ...)
  This primitive is used to create a coroutine. It allocates a new hook filled with the continuation of the `coroutine` form. Then the `f` function is applied on the hook. When `f` returns, the continuation currently held in the hook is invoked on the returned value.
- (`escape hook val`)
  This primitive allows a coroutine to exit, passing control to another coroutine. It invokes the continuation held in `hook` on `val`. This means that `val` is returned

as the value of the `coroutine` or `suspend/resume!` form that was the last to fill the hook.

- (`suspend/resume! hook val`)
  This primitive allows the suspension of the current coroutine, resuming another previously suspended coroutine. Therefore this is a kind of explicit context switch mechanism between coroutines. It exchanges the current continuation with the continuation held in `hook`, and invokes this latter on `val`.

The primitives `coroutine` and `suspend/resume!` are used to capture continuations just as `call/cc` in Scheme. However, continuations can't be obtained as first-class objects, because there is no legal operation reading the hook contents directly. Yet, the original first class continuations are recovered by defining:

```
(define (call/cc fun)
  (coroutine
    (lambda (hook)
      (fun
        (lambda (value)
          (escape hook value))))))
```

The inner lambda that is passed to `fun` emulates the Scheme continuations. It is actually a first-class function because closures are first-class in Scheme. Note that the continuation held in a hook is not lost when `escape` is used, so it can be reinvoked. In this way, multiple returns from function applications are also recovered.

Nevertheless, just using this newly defined `call/cc` gives no performance advantages over the Scheme first-class continuations. We will see that an efficient implementation can be conceived for applications creating a moderate number of coroutines but heavily exchanging control, as in the following solution for the *same-fringe* problem:

```
;;;a leaf reader
(define (make-walker tree)
  (coroutine
    (lambda (hook)
      ;;a recursive traversal
      (define (walk tree)
        (cond
          ((not (pair? tree))
            (suspend/resume!
              hook tree))
          (else
            (walk (car tree))
            (walk (cdr tree)))))
      ;;returns the hook
      ;;to the client
      (suspend/resume! hook hook)
      ;;starts the traversal
      (walk tree)
      ;;signals the end
      'end )))
```

```
;;;The comparator
(define (same-fringe tree-a tree-b)
  ;;starts the two leaf readers
  (define hook-a
    (make-walker tree-a))
  (define hook-b
    (make-walker tree-b))
  ;;loops on the leaves
  (let loop ()
    (let ((leaf-a (suspend/resume!
                    hook-a 'void))
          (leaf-b (suspend/resume!
                    hook-b 'void)))
      (cond
        ((not (eq? leaf-a leaf-b))
              #f)
        ((eq? leaf-a 'end)
              #t)
        (else
              (loop))))))
```

4 Implementing second class continuations

Let us consider a Scheme implementation passing arguments in registers and allocating a fixed size frame at function entry. This frame is allocated in a special heap controlled by the frame memory manager presented in next section. A frame contains the following fields:

- `tag`: A frame identifier used by the frame memory manager. This tag can be a pointer to a structure containing the frame layout.
- `retaddr`: The caller return address.
- `oldfp`: The caller frame address. The fields `retaddr` and `oldfp` represent the implicit continuation passed to every function.
- Some optional static frame pointers: Present only when the function accesses variables located in lexically enclosing functions.
- Some variables: The programmer defined variables, the function arguments and some intermediate values, which are held in registers initially, but need to be saved when a function call uses some of those registers and also at register exhaustion.

Upon function entry, a frame is allocated and initialized with the caller information. The frame address is placed in a dedicated register named *current frame pointer* or simply `fp`. At return, `fp` is restored with the caller frame address and a jump to the caller return address is done. Since a frame can still be useful even after function return, it can't be freed as easily as in a stack implementation. When there is no more memory for allocating frames, the frame memory manager reorganizes the heap by pruning frames that are no longer reachable from the current frame pointer or a continuation held in a hook.

A hook is a structure allocated in the general heap. It contains a `tag` identifier used by the memory manager and a field named `cfp` which is a pointer to a *capture frame* structure. A capture frame is a special frame created at a continuation capture for storing the information needed to invoke that continuation. Excepting a hook, there is no other first-class object pointing to a capture frame. A capture frame is allocated in the special frame heap and contains the following fields:

- `tag`: A capture frame identifier.
- `retaddr`: The return address to jump to when invoking the continuation.
- `oldfp`: The frame address of the function being suspended.
- `hook`: The address of the hook involved in the continuation capture and which will be linked to this capture frame.
- `nextcfp`: This pointer is used by the frame memory manager.

Figure 1 shows the linking between a hook, a normal frame and a capture frame.

Using these hook and capture frame structures, the coroutine primitives are implemented as follows:

- `(coroutine f)`: A capture frame `cfp` is allocated with `tag`, `retaddr` and `oldfp` filled as upon normal function entry. Next, `hook` is allocated to hold the continuation of the `coroutine` form. Then the following code is executed:

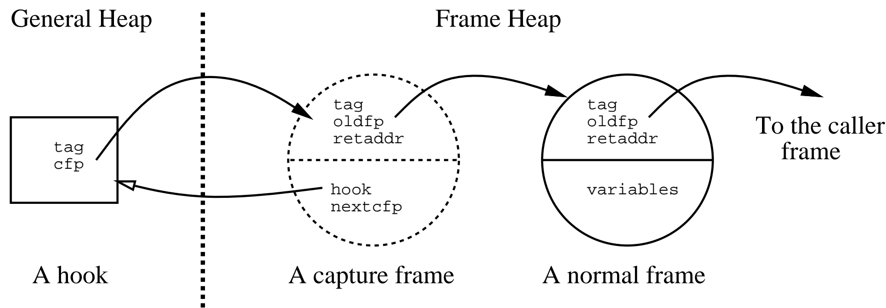

Fig. 1. Hook and frame linking.

```
cfp→hook= hook              ;; The capture frame and the hook
hook→cfp= cfp               ;; are made to point at each other.
cfp→nextcfp= listcfp        ;; The capture frame is chained in a list,
listcfp= cfp                ;; for reasons which we will explain later.
fp= cfp                     ;; The capture frame becomes the current
escape(hook, f(hook))       ;; frame and f is invoked.
```

Note that upon normal return of `f`, this form does an escape through the current hook contents.

- (`escape hook val`): A normal return is done as if the current frame was the capture frame referenced by `hook`.

```
fp= hook→cfp
return val
```

- (`suspend/resume! hook val`): A capture frame `cfp` is allocated with `tag`, `retaddr` and `oldfp` filled as upon normal function entry. Then the following code is executed:

```
fp= hook→cfp                ;; The capture frame in hook
                            ;; becomes the current frame.
hook→cfp= cfp               ;; Then hook is linked to cfp.
cfp→nextcfp= listcfp        ;; The capture frame is chained
listcfp= cfp                ;; in a list, as in coroutine.
return val                  ;; A normal return is made from
                            ;; the new current frame.
```

Figure 2 shows the frame and hook chaining for an example of function call tree. Frames have been enumerated according to allocation order. While working with frame 1, a coroutine is created, so the hook H and the capture frame 2 are allocated. Then the frame 3 is allocated for the function starting that coroutine. Next, using `suspend/resume!` through the hook H, that coroutine is suspended and the work with frame 1 resumed, so the capture frame 4 is allocated. A function call allocates frame 5 where a `suspend/resume!` through hook H creates the capture frame 6 and resumes the work with frame 3. Another function call allocates the frame 7 from where an escape through hook H is done, resuming the work with frame 5. Finally

a normal return resumes the work with frame 1 from where a last function is called allocating frame 8.

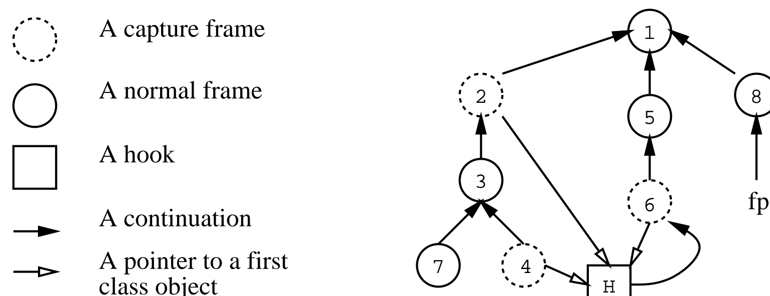

Fig. 2. An example of function call tree.

Initially, the hook H has been linked to frame 2, but the two successive `suspend/resume!` operations link it to frame 4 and then frame 6. Note that frame 7 hasn't been captured, so its memory is available for new allocation. In the same way, the capture frames 2 and 4 aren't reachable from any hook, so the memory taken by frames 2 and 4 and then frame 3 is also available.

5 An efficient memory manager for frames

We split memory management into two almost independent garbage collectors. The first is the general garbage collector managing first class objects such as cons cells, symbols, vectors, etc. and especially hooks. The second is the frame only memory manager —including capture frames— which is presented in this section. This frame memory manager is a simplification of the generational *Stop and Copy* garbage collector described in [Nakajima 88] and [Appel 89].

Frame allocation is implemented as follows. At the beginning of a cycle, there is an empty buffer which we will name the primary buffer. Two registers `hp_limit` and `hp` point at the start and the end of this buffer. A frame is allocated by subtracting its size from `hp` and comparing the new `hp` against the `hp_limit` register to test the buffer overflow. It is important to note that there is no need to initialize frames, instead the primary buffer is cleaned of dangling references by initializing it to zero or nil after a general garbage collection.

When the primary buffer overflows, it holds the frames for the complete function call tree from the beginning of the cycle. As stated in the previous section, some branches of this tree are unreachable, so the reachable ones are appended to another buffer which we will name the secondary buffer. Then a new cycle begins with an empty primary buffer.

The reachable frames are firstly, those captured by the current frame which signaled the overflow, and secondly, frames that have been captured by a continuation held in a hook. However, for the latter, it has been necessary to create a capture

frame in the primary buffer from where they are reachable. Therefore `coroutine` and `suspend/resume!` chain the capture frames that they create in a list which we will name the primary capture frame list or simply `listcfp`. In addition, some of these capture frames are no longer referenced by a hook, because their initial hook has been used to capture another continuation, so they are considered unreachable unless another continuation captures them.

We outline a simple copying collector to transfer frames to the secondary buffer. We say that this collector prunes frame trees.

1. For each capture frame C in `listcfp`:
   (a) If C points to a hook no longer linked to C, continue with the next capture frame in `listcfp`.
   (b) Reverse the dynamic chain obtained from C by following the `oldfp` field as far as a frame located in the secondary buffer or a frame that has been marked as already transferred.
   (c) For each frame F in this new chain:
      i. Make a copy of F in the secondary buffer. This copy will be named F'. The size of F is determined from the `tag` field.
      ii. Link `oldfp` in F' to the copy of the caller frame which is just the previously transferred frame.
      iii. Set a mark in the `tag` field in F indicating that F has been transferred.
      iv. Link `oldfp` in F to the address of F'.
      v. If F has some static pointers, since the referenced frames are in the dynamic chain, they have already been transferred, so relink any static pointer to its new address. This address is found in the `oldfp` field of the referenced frame.
   (d) Let C' be the copy of C. C' points to a hook having the `cfp` field linked to C. Link `cfp` to C'. Then chain C' into a list which we will name the secondary capture frame list.
2. Transfer in a similar way the dynamic chain obtained from the current frame pointer.
3. Set `listcfp` to the empty list.

Afterwards, the execution must be resumed with a primary buffer reduced to the size of the remaining memory in the secondary buffer. Thus, when the primary buffer overflows again, all new frames are guaranteed to find room in the secondary buffer, even when all of them are reachable. When this frame pruning is triggered after a primary buffer overflow, we call it a *minor pruning*.

If the primary buffer becomes too small —assume a quarter of its initial size— make a *major pruning*. A major pruning exchanges the primary and secondary buffers and the primary and secondary capture frame lists, and then does a normal pruning. Most old frames transferred to the secondary buffer aren't reachable, so they won't be recopied and the secondary buffer will regain a reasonable size.

Figure 3 shows the primary and secondary buffers before pruning the tree of Fig. 2. Frame 1 is the only frame already located in the secondary buffer. Figure 4 shows the buffers once the frames reachable through hook H have been transferred. Finally, Fig. 5 shows the buffers once the pruning has finished.

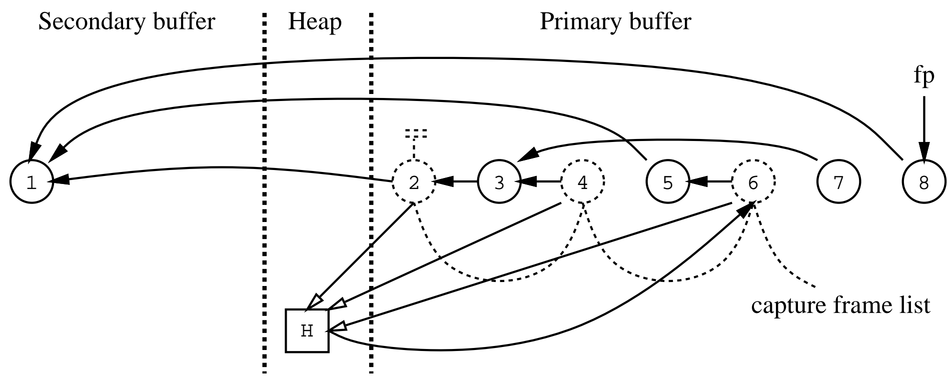

Fig. 3. Frame tree before pruning.

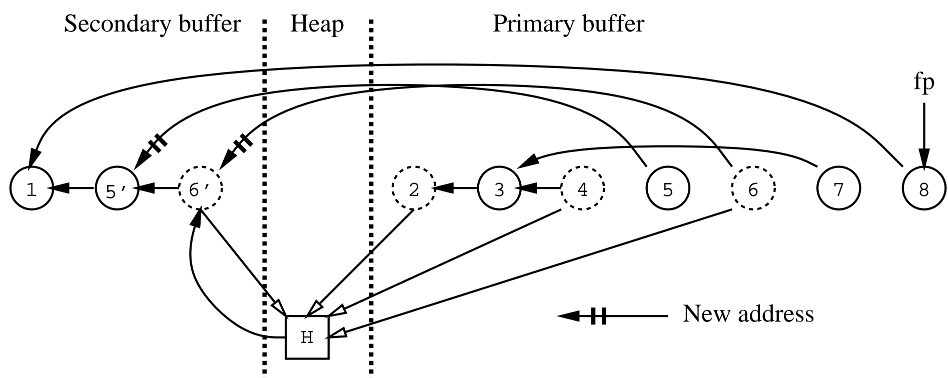

Fig. 4. Buffer contents after transferring frames reachable through hook H.

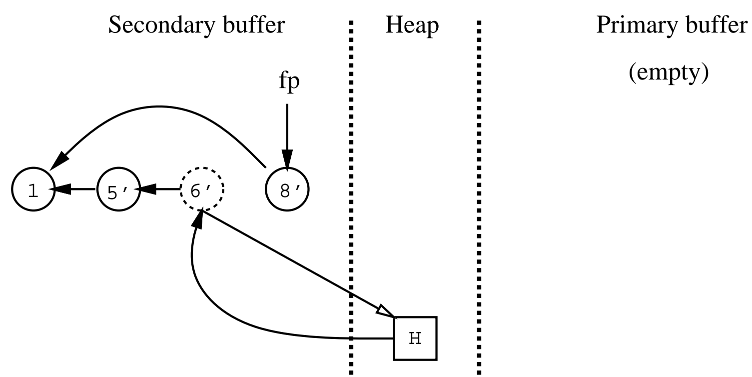

Fig. 5. Final pruned frame tree.

Having an unlimited extent, a hook might become garbage for the rest of the execution. Its captured frames will therefore appear to be reachable forever to the frame memory manager. These useless frames will fill the secondary buffer increasing the pruning frequency and so degrading performance. Hence, if after a major pruning the secondary buffer is filled up to a given percentage, a general garbage collection must be triggered to discard the unreferenced hooks. However, this situation should occur rarely because it is the normal heap exhaustion which will trigger the collection.

6 Synchronization between first-class object and frame memory management

During a general garbage collection, a subtle synchronization with the frame memory manager must be done to recover the frame space captured by garbage hooks. To achieve this goal, while doing the general garbage collection, a special major pruning is carried out simultaneously as follows:

1. Make a minor pruning to free the primary buffer.
2. Exchange primary and secondary buffers.
3. Set the secondary capture frame list to empty.
4. Transfer the frames reachable from the current frame to the secondary buffer. The pointers held in those frames are roots for the general garbage collector.
5. Start the general garbage collector.
6. For each hook proved reachable by the general garbage collector:
   (a) Transfer the frames captured by the hook to the secondary buffer.
   (b) Chain the copy of the associated capture frame in the secondary capture frame list.
   (c) The pointers held in the transferred frames are roots for the general garbage collector.

Note that the method used by the general garbage collector doesn't matter. It could be a *Stop and Copy* or a *Mark and Sweep* collector with or without generations, etc.

Our frame memory manager and the general memory manager, as a whole, can be seen as a generational garbage collector [Lieberman & Hewitt 83], with the first two generations reserved for frames only. The complexity associated with generational memory management comes from the need to trace pointers from older generations to newer generations. These pointers are the result of object mutating operations such as `set-car!`, `vector-set!`, etc. However, frame chaining can't be altered in the frame heap, so pointers from secondary buffer to primary buffer can't exist and therefore no tracing is needed. Yet, coroutine suspension can link a hook —in an old generation— with a newer frame, hence the necessity to introduce capture frames which just serve to trace hook mutations.

7 Performance analysis for applications lacking coroutines

In this section we evaluate the overhead that the memory organization described in previous sections carries to applications not using coroutines. To measure this overhead we have implemented it in F1 [Seniak 91]. F1 is a compiler for a small lisp, generating assembler code for Sparcs. It uses the 31 Sparc registers as much as possible, but without calling on window registers. F1 doesn't treat floating point numbers, so no test is needed in integer arithmetic operations.

We show in table 1 the timings for the Gabriel benchmarks [Gabriel 85]. Timings include our heap organization as well as various stack implementations which test or don't test overflow and chain or don't chain frames. We include also the performances of a mixed stack/heap strategy which we explain in the next section. The measurements have been done on a SUN/670MP, having a 64 Kb cache.

To measure the overhead associated with tagging, we added the tags to the frame chaining stack organization, then this overhead was the additional execution time. To measure the overhead associated with frame pruning, we doubled the work by transferring surviving frames to a third intermediate buffer while pruning, then another pruning transferred the same frames to the secondary buffer. The pruning overhead was the difference with respect to our heap organization. Having the overhead of frame chaining, tagging and pruning, the remaining unexplained overhead was due to the locality loss in memory access.

Therefore the proposed organization has an overhead of 18% when compared with a stack implementation testing the overflow. However with the optimizations described in the next section, we reduce that overhead to a 11%.

The surprisingly small overhead of frame pruning is explained because the mean lifetime of frames is very short in conventional applications. The frames surviving to a minor pruning are those that are reachable from the current frame, excepting those that are already in the secondary buffer. Therefore, the number of transferred frames is the depth of the call tree at primary buffer overflow, less the minimal depth reached during the cycle. However, considering the locality of function call depth from which Sparc register windows are inspired, this number is very small. In fact, we have measured a 1 to 2% of frames surviving to a minor pruning, and a 2 to 5% surviving to a major pruning.

The locality loss is the main penalty for this heap organization. A normal stack organization presents a high degree of locality, explained also by the locality of function call depth. However, our organization allocates frames sequentially, flushing cache lines at almost every function call. In fact the 5.2% was obtained solely when the primary buffer was limited to a size by 16 or 32 Kb, to give an opportunity to the 64 kb cache to hold it completely, otherwise the overhead was greater.

8 Optimizing performances of frame allocation

The following minor variation is inspired from the stack/heap strategy described in [Clinger *et al* 88].

Upon normal function entry, after frame allocation, the `fp` and `hp` registers have the same value. If this still holds at return, no capture frame could be allocated,

so the frame memory can be reused safely. Therefore, at return, the `fp` and `hp` are compared and when they are equal, the allocated frame is freed by adding its size to the `hp` register. Moreover, when there is no captured continuations, a function calling another function will get the `hp` register with the same value that it had before the call, so it will also free its frame at return. Thus, frames will be pushed and popped in the primary buffer, just as in a stack, and the application will exhibit the locality of a stack organization.

During a continuation capture, the `hp` register is adjusted to allocate a capture frame. When that continuation is invoked, the `hp` register is not restored, so its further comparison against `fp` will fail, and all frames allocated before the continuation capture will not be freed. In this way, the frames captured by a continuation are guaranteed not to be reused for new allocation. Frames allocated after the capture frame will continue with the normal push and pop discipline.

Therefore, in normal applications, there is a gain in the locality of memory access, but there is also a loss in performing the test at function return. Although this test is useless when coroutines are exploited intensively, we have adopted it, because we desired a minimal penalty for normal applications and the measurements had shown that the gains were greater than the losses.

Another optimization adopted is the suppressing of frame tagging. In fact, the tags can be placed at primary buffer overflow for the reachable frames only. The tags can be deduced from the return address by using a binary tree, a hash table or, in some architectures, just including it in the code around the return address. The performance of an implementation with the stack/heap optimization and tag suppression is shown in table 1.

Finally, other optimizations are possible, even though we didn't adopt them. First, the overflow test at function entry can be suppressed by placing the frame heap in the stack space of a Unix process and organizing it as in Fig. 6. In this way, the test is done at continuation captures only, while at deep recursion the primary buffer grows automatically. Second, the return test can also be suppressed in functions which are known not to capture continuations by inspecting the static function call tree. In addition, those functions don't need the frame chaining. Third and last, established optimizations such as function integration and inlining can be applied to further reduce the call/return overhead.

9 Performance analysis for the same-fringe problem

Tables 2, 3 and 4 show the performances obtained by three solutions for the *same-fringe* problem. Each version compares 10 times two trees containing 100,000 cons cells.

The first solution is based on coroutines, so it does no allocation in the general heap, instead it captures a continuation at every leaf. Although the stack/heap optimization is present, it is useless, and therefore there is a low locality in memory access. Table 2 shows performances when varying the primary buffer size and the total frame heap size.

The second solution uses `call/cc` to emulate the coroutine context switch. For implementing `call/cc` we used a variation of the stack/heap strategy where a general

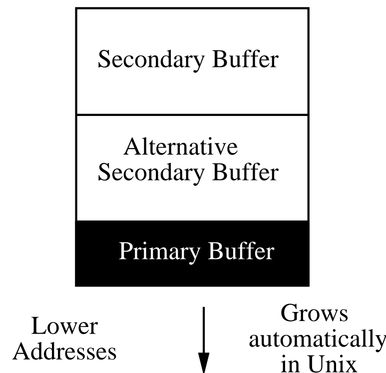

Fig. 6. A buffer organization allowing the suppression of the overflow test and simplifying the first ancestor search when implementing shallow binding. At a major pruning, frames are transferred from the secondary buffer to the alternative secondary buffer or vice versa.

garbage collection is triggered immediately after a frame heap overflow. We are constrained to do so because the first-class status of Scheme continuations exclude any frame pruning without proving that the continuation which captured a frame is not referenced by another first-class object.

The third solution flattens the trees, i.e. it chains the leaves of each tree into two lists before comparing them, resulting in a high general garbage collection activity. We used a Stop and Copy collector with no generations.

The measurements show that the version using `call/cc` is far the slowest. They show also that there is a performance crosspoint between the coroutine and the tree flattening solutions when objects survive in average one collection cycle in the tree flattening version. Therefore the latter will win especially when coupled to a generational garbage collector where objects rarely survive to one generation.

However, we think that coroutines must not be seen as a means to speed up applications. Instead, they are a powerful abstraction tool which can greatly simplify programming. The aim of presenting these measurements is to show that our coroutines aren't expensive even when used intensively as in *same-fringe*. In fact, in the coroutine version, *same-fringe* reaches a rate of 430,000 `suspend/resume!` per second. Yet, if *same-fringe* is considered as a subpart of a more complex system, the coroutines solution could win when a generational approach is not desired, because the programming paradigm involves too much object mutations, such as in object oriented systems.

10 Extensions

In this section we discuss some possible extensions to the memory organization described previously. We start by introducing a hint to implement shallow binding for dynamic variables; next, we discuss a way to treat dynamic escapes efficiently; and finally, we consider allocating dynamic objects in the frame heap.

Dynamic variables have been traditionally implemented in high-performance Lisps by using shallow binding, because the time needed to create, access and delete a dynamic variable is constant. However, combining shallow binding with coroutines or first-class continuations introduces a subtle complication. When transferring control between coroutines, the dynamic environment must first be unwound from the current frame up to a common ancestor with the target frame, and then, rewound down to the target frame.

This overhead in restoring dynamic environments can discourage language designers to add dynamic variables, because even when these variables are not used, coroutine users must pay at least the cost of finding the common ancestor to discover that there is no dynamic environment to restore. This search can be accelerated by chaining the frames containing dynamic variables in a special list. Yet, adding a single dynamic variable to the program, introduces an additional overhead in any control exchange between coroutines.

Therefore we point out an interesting property for the buffer organization of Fig. 6 when using the frame pruning of Sect. 5. For every frame f_1 pointing to a frame f_2, the following holds:

$$\text{address}(f_1) < \text{address}(f_2)$$

Hence, finding the common ancestor between two frames is as easy as unwinding the two dynamic chains up step by step, alternating in such a way that the chain containing the lower address frame is unwound first. The unwinding stops when the same frame is found. Thus, when there is no dynamic environment to be restored, the overhead associated with the search of the common ancestor is reduced to a single test.

Another desirable extension is a way to treat *dynamic extent continuations* efficiently. These continuations are useful to implement fast *escapes* such as `longjmp` in C. A dynamic extent continuation can only be invoked by a function that is a child of the function which captured that continuation. The capture of a dynamic extent continuation is implemented by allocating a hook and a *dynamic capture frame*, much as `coroutine` is implemented. A dynamic capture frame is a capture frame, but the fact that the former is referenced by a hook isn't enough to consider that frame reachable as is the case for the latter: the former must also be referenced by a reachable frame. Thus, there is no need to trigger an expensive general garbage collector to recover frames captured by a dynamic extent continuation. Detecting the illegal use of a dynamic continuation is achieved as follows: at a frame pruning, when a hook is linked to a dynamic capture frame considered no longer reachable, that hook is redirected to a special capture frame containing an error handler in its return address.

Finally, in a stack based organization, dynamic extent objects can be allocated efficiently in the stack. In our heap organization, such objects can also be treated efficiently, because the compiler can be modified to include layout information to be used at frame pruning. This information must describe where to find pointers to dynamic objects in the frames of functions allocating, or receiving in arguments, such objects.

11 Conclusions

In conceiving our memory organization for frames, we were inspired from [Appel 87] which states that garbage collection can be faster than stack allocation if very large heaps are coupled with a Stop and Copy collector. Although the original idea is not practical with current memory configurations, we found that it was reasonable when applied to Spaghetti stacks [Bobrow & Wegbreit 73], because stack memory requirements are much smaller than heap memory requirements. Then, we realized that adding generations to frame pruning was easy, because pointers from older frames to newer frames can't exist. Finally, experimentation established that frames have a very short life time, reaching to a point where frame pruning is almost costless. Therefore, large frame heaps are not recommended because the smaller ones are more efficient in presence of memory caches.

In this paper we concentrated on proving that this memory organization can be used effectively to implement a simple class of coroutines. However, we are conscious that the primitives we have introduced don't have all the desired power required from coroutines. Yet, they don't extract the full power of the memory organization. For example, additional power can be obtained by including two new primitives to capture a continuation in a previously existing hook and to displace a continuation from one hook to a second hook. Such additions would not affect the performances of the frame pruner.

Acknowledgements

The author wishes to thank Nitsan Séniak for his valuable explanations about the working of his F1 compiler, Christian Queinnec for interesting discussions concerning the semantics of continuations and helpful remarks on the writing of this paper, and David De Roure for precious improvements brought to this paper.

References

[Appel 87] Andrew W. Appel: "Garbage Collection Can Be Faster Than Stack Allocation," *Information Processing Letters* 25, 1987, 275-279.

[Appel 89] Andrew W. Appel: "Simple Generational Garbage Collection and Fast Allocation", *Software-Practice and Experience*, 19(2), February 1989, 171-183.

[Bobrow & Wegbreit 73] Daniel G. Bobrow and Ben Wegbreit: "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM*, 16(10), October 1973, 591-603.

[Clinger *et al* 88] William D. Clinger, Anne H. Hartheimer and Eric M. Ost: "Implementation Strategies for Continuations," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, 124-131.

[Gabriel 85] Richard P. Gabriel: *Performance and Evaluation of Lisp Systems*, the MIT Press, 1985.

[Haynes *et al* 86] Christopher T. Haynes, Daniel P. Friedman and Mitchell Wand: "Obtaining Coroutines with Continuations", *Computer Languages*, 11(3/4), 1986, 143-153.

[Hieb *et al* 90] Robert Hieb, R. Kent Dybvig and Carl Bruggeman: "Representing Control in Presence of First-Class Continuations," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20-22, 1990, 66-77.

[Lieberman & Hewitt 83] Henry Lieberman and Carl Hewitt: "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM*, 26(6), June 1983, 419-429.

[Nakajima 88] Katsuto Nakajima: "Piling GC — Efficient Garbage Collection for AI Languages —," *Parallel Processing*, M. Cosnard, M. H. Barton and M. Vanneschi (Editors), Elsevier Science Publishers B.V. (North Holland), IFIP, 1988, 201-204.

[Rees & Clinger 86] Jonathan A. Rees and William Clinger, eds.: "The Revised[3] Report on the Algorithmic Language Scheme," *SIGPLAN Notices*, 21(12), December 1986.

[Seniak 91] Nitsan Séniak: *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels*, Thèse de Doctorat de l'Université Paris 6, October 1991.

Appendix: Defining coroutines from Scheme continuations

In this appendix we give a semantics for the primitives presented in Sect. 3. The formal behavior is obtained by defining each primitive from Scheme continuations.

```
;;; Coroutine creation
(define (coroutine fun)
  (call/cc
    (lambda (k)
      (fun (make-hook k)))))

;;; Escaping
(define (escape hook val)
  ((hook-ref hook) val))

;;; Suspension and resumption
(define (suspend/resume! hook val)
  (call/cc
    (lambda (k)
      (let ((old-k (hook-ref hook)))
        (hook-set! hook k)
        (old-k val)))))
```

```
;;; The hook abstraction
(define (make-hook k)
  (vector k))

(define (hook-ref hook)
  (vector-ref hook 0))

(define (hook-set! hook k)
  (vector-set! hook 0 k))
```

Table 1. Performances of different implementations for frame allocation. The first three columns correspond to the execution time of three stack implementations. The first one does no stack overflow test nor frame chaining, the second adds stack overflow test and the third adds both. We consider the second implementation as the "normal" stack implementation. The following four columns show the execution time of our heap implementation and the relative overhead associated with tagging, frame pruning and locality loss, compared to the normal stack implementation. Finally, the last column shows the performances of our organization with the stack/heap optimization and tag elimination. All execution times are expressed in seconds and the percentage appearing under each of them is the relative overhead of the corresponding implementation, compared with the normal stack implementation. Note that the sum of the relative overheads associated with frame chaining, tagging, pruning and locality loss is the relative overhead of the heap implementation.

Gabriel benchmark timings (in seconds)								
Benchmark name	Stack			Heap				Stack/ Heap
	-ovf test	normal	+frame chaining	total	tag	pruning	loc. loss	
Puzzle	0.822	0.823	0.836	0.890				0.883
	-0.1%		1.6%	8.1%	1.0%	0.1%	5.5%	7.3%
Boyer	0.885	0.930	1.058	1.210				1.102
	-4.8%		13.7%	30.1%	5.6%	4.3%	6.5%	18.5%
Destru	1.700	1.730	1.770	1.835				1.790
	-1.7%		2.3%	6.1%	1.4%	0.3%	2.0%	3.5%
Browse	1.590	1.625	1.700	1.840				1.765
	-2.2%		4.6%	13.2%	3.4%	2.5%	2.8%	8.6%
Div-rec	0.145	0.155	0.170	0.202				0.180
	-6.5%		9.7%	30.6%	6.5%	6.5%	8.1%	16.1%
Div-iter	0.140	0.140	0.140	0.142				0.140
	0.0%		0.0%	1.8%	0.0%	0.0%	1.8%	0.0%
Deriv	0.290	0.298	0.315	0.345				0.320
	-2.5%		5.9%	16.0%	3.4%	0.0%	6.7%	7.6%
Dderiv	0.338	0.345	0.372	0.415				0.388
	-2.2%		8.0%	20.3%	5.1%	2.9%	4.3%	12.3%
Triangle	8.895	9.190	9.935	10.810				10.420
	-3.2%		8.1%	17.6%	4.8%	1.1%	3.5%	13.4%
Traverse	5.730	5.875	6.355	7.200				6.495
	-2.5%		8.2%	22.6%	5.2%	0.3%	8.9%	10.6%
Tak	0.043	0.046	0.053	0.061				0.055
	-5.2%		15.8%	33.3%	7.8%	2.8%	6.9%	21.0%
Average	-2.8%		7.1%	**18.2%**	4.0%	1.9%	5.2%	**10.8%**

Table 2. Performances of a *same-fringe* solution based on coroutines. The columns show the total size of the frame heap, the primary buffer size, the total amount of memory allocated for frames, the percentage of frames surviving to a minor pruning, the number of minor and major prunings, and finally, the execution time using the stack/heap implementation.

Same-fringe with coroutines						
frame heap size (KB)	primary buffer size (KB)	frames allocated (KB)	copied frames	minor prunings	major prunings	execution time (secs.)
128	8	97735	1.77%	12997	49	9.4
128	16	97735	1.57%	6300	32	9.3
128	32	97735	0.87%	3101	26	9.4
256	64	97735	0.47%	1538	7	9.6
512	128	97735	0.25%	766	1	10.9

Table 3. Performances of a *same-fringe* solution with `call/cc`. The columns show the total heap size, the total amount of memory allocated (mainly formed of frames and continuations), the memory copied during garbage collection (including the trees), the number of garbage collections and the execution time using the stack/heap variation where the general garbage collector is triggered immediately after a frame heap overflow.

Same-fringe with `call/cc`				
heap size (KB)	allocated objects (KB)	copied objects (KB)	number of GCs	execution time (secs.)
7000	96117	189916	81	81.6
10000	96117	82063	35	43.4
12000	96117	58616	25	35.1
15000	96117	42204	18	29.3
20000	96117	28135	12	24.3

Table 4. Performances of a *same-fringe* solution with tree flattening. The columns show the total heap size for a Stop and Copy collector, the memory allocation in cons cells, the total memory copied during garbage collection (including the two trees), the execution time with a stack implementation and the execution time with our stack/heap implementation.

Same-fringe with tree flattening					
heap size (KB)	allocated objects (KB)	copied objects (KB)	number of GCs	execution time (secs.)	
				stack	stack/heap
10000	25781	23905	9	13.6	14.0
12000	25781	32905	9	16.8	17.1
15000	25781	11249	4	8.9	9.3
20000	25781	8759	3	8.2	8.4