Bypassing of Channels in Eden*

Ulrike Klusik[1], Ricardo Peña[2], and Clara Segura[2]

[1] Philipps–Universität Marburg, D-35032 Marburg, Germany
Phone# +49-6421-281521 ; Fax# +49-6421-285419
e-mail: klusik@Mathematik.Uni-Marburg.de

[2] Universidad Complutense de Madrid, E-28040 Madrid, Spain
Phone# +34-91-3944313; Fax# +34-91-3944602
e-mail: {ricardop,csegura}@eucmax.sim.ucm.es

Abstract. We describe *automatic bypassing*, a desirable optimization of Eden's implementation aimed at reducing the number of messages and/or threads at runtime. Eden [BLOMP97] extends the lazy functional language Haskell with a set of *coordination* features, aimed to express parallel algorithms. These include *process abstractions* (or process schemes) and *process instantiations* (or applications of a process scheme to actual inputs).
When a new process is instantiated, their input and output channels are connected to its parent process. This implies that, in principle, only tree–like process topologies can be created. But the aimed topology may not be hierarchical (e.g. pipelines, grids, etc.). It is desirable to be able to connect every producer to its actual consumer, trying to avoid the intermediate processes frequently used only to set up the topology.
The strategy consists of a combination of compile time analysis and runtime support. Both are explained in detail. Also, the savings expected with the proposed strategy are commented.

1 Introduction

The parallel functional programming language Eden [BLOMP97], [KOMP98] extends the lazy functional language Haskell by syntactic constructs to explicitly define processes. Eden is implemented by modifying the Glasgow Haskell compiler GHC [JHH+93]. In [BKL98], more details are given.

Processes in Eden are dinamically instantiated while executing recursive functions and/or process definitions. When a new process is created, their channels are connected to its parent process. The parent is responsible for feeding child's input channels with values and for receiving values from child's output channels. This implies that, in principle, only hierarchical process topologies can be created. Frecuently, it is the case that the parent process just copies the values received from one child to an input channel of another child that, in turn, forwards these values to another process, and so on. It is a desirable optimization to detect this situation to be able to eliminate intermediate processes in the transmissions, i.e. to directly connect producers to consumers. This would save a lot of messages and much useless computations at runtime.

We call the optimization *automatic bypassing*, which consists of a combination of compile time analysis and runtime support. At compile time, the forwarding of inputs to outputs within a process definition and between process instantiations is detected, and then the program is annotated so that the local 'forwards' are known at runtime. At that moment, a global bypassing view is generated by defining a communication protocol that introduces the producer to its real consumer.

The organization of the paper is as follows: in Section 2 we introduce the Eden features relevant to bypassing and explain the bypassing problem in detail. In Section 3 Eden abstract syntax, before and after annotations, is presented. Section 4 formally defines the compile time analysis and applies it to a simple example. Section 5 explains the bypassing protocol and the support given to it by the runtime system (RTS). Finally, Section 6 gives an account of the expected savings and summarizes the state of the implementation.

* Work partially supported by the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

2 Eden and bypassing

Functional languages distinguish between function definitions and function applications. Much in the same spirit, Eden offers *process abstractions*, i.e. abstract schemes for process behaviour, and *process instantiations* for the actual creation of processes. Process abstractions have a polymorphic type `Process a b` and can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Process abstractions are first class values, so they can be used as parameters or results of functions. A process abstraction mapping input variables $x_1, \ldots, x_n$ to output expressions $exp_1, \ldots, exp_k$ can be specified by the following expression:

$$\texttt{process}\ (x_1, \ldots, x_n)\ \texttt{->}\ (exp_1, \ldots, exp_k)$$
$$\texttt{where}\ equation_1 \ldots equation_r$$

The output expressions can reference the input variables, as well as the auxiliary functions and common subexpressions defined in the optional `where` part. A process communicates to other processes through tuples of channels and arbitrary data structures of channels. In this paper we restrict ourselves to tuples.

A *process instantiation* is achieved by using the predefined infix operator `(#)::Process a b -> a -> b` in the following way: $(y_1, \ldots, y_k) = p \# (exp_1, \ldots, exp_n)$.

We will refer to the new instantiated process as the *child* process, while the process where the instantiation takes place, will be called the *parent*. The process abstraction bound to p is applied to a tuple of input expressions, yielding a tuple of output variables. The child process uses k independent threads of control in order to produce these outputs. Correspondingly, the parent process creates n additional threads for evaluating $exp_1, \ldots, exp_n$. In both cases, value production is done eagerly. Communication is unidirectional, from one producer to exactly one consumer. In order to provide control over where expressions are evaluated, only fully evaluated data objects are communicated. Lists are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to normal form and then transmitted.

To illustrate the bypassing problem, here are the definition and the instantiation of a pipeline:

```
pipe :: [Process a a] -> Process a a
pipe [p] = p
pipe (p:ps) = process z -> pipe ps # (p # z)
y = pipe [p1,p2,p3] # x
```

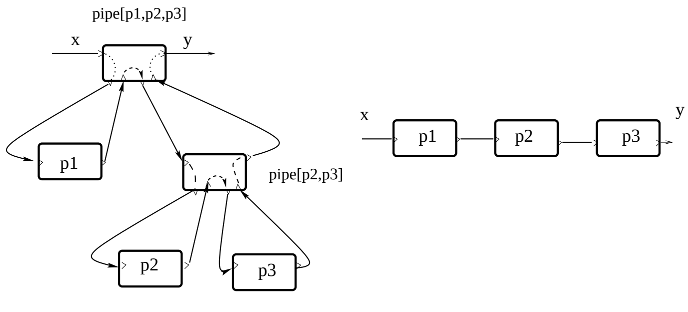


Fig. 1. Created and desired topology for a pipeline

Eden's current implementation [KOMP98] unfolds at runtime the recursive definition and generates the process topology of Figure 1 (left). It can be seen that this structure is far from what is desirable —the one in Figure 1 (right)— because processes are created not only for `p1, p2` and `p3`, but also for `pipe [p1,p2,p3]` and `pipe [p2,p3]`, which only forward data from their input to their children and from them to their output. Channels are connected in a hierarchical way by following the unfolding of the recursive definition.

Automatic bypassing will detect such situations by using a combination of compile time analysis and of runtime protocol, in order to connect producer to consumer directly, as in Figure 1 (right). Nevertheless, with our approach the additional processes will still be instantiated, but they will terminate after creating their children processes as they neither produce nor consume data.

This example illustrates two of the three possible kinds of bypassing: *bypassing between siblings* and *bypassing between generations*. There is a third kind: when an input channel value is directly copied to exactly one output channel. We call this *bypassing between ancestors*. The analysis will detect the three situations.

In Figure 2 the different kinds of local forwards are shown: the descendant forwards correspond to bypassing between siblings, the ancestor forwards correspond to bypassing between ancestors, and the ascending and descending forwards correspond to bypassing between generations.

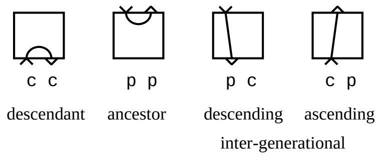


Fig. 2. Forward classes

3 CoreEden and Annotated CoreEden

Currently in Core[1], process abstractions and instantiations are hidden inside predefined functions. In order to do bypassing analysis we need to make input and output channels explicit. So, we introduce a new intermediate language called CoreEden, which is an extension of Core. The basic extensions are the introduction of a new expression for process abstractions, and a new type of binding which we call **recpar** where usual Core bindings and new ones called process instantiation bindings appear. The latter have the form $channels = var \,\#\#\, channels$. The (sugared) CoreEden abstract syntax is depicted in Figure 3, where a $*$ marks the constructs added to Core. Initially there are no **bypass** clauses. They are introduced after the analysis.

The idea is first to translate from Core to CoreEden; then to do the bypassing analysis, producing an annotated CoreEden program and finally to translate back from CoreEden to Core to go on with the rest of the compilation. Inside CoreEden there are also some transformations, called flattening and untupling transformations, which try to collect as much information related to the process abstractions and instantiations as we can. In this paper we concentrate in the bypassing analysis. The other aspects will be shown elsewhere. The analysis decorates process abstractions and **recpar** bindings with bypassing annotations of the form **bypass** *channels*. In the next section we will use Exp as the type of a CoreEden expression, and $DExp$ as the type of an annotated expression (similarly, $Binds$ and $DBinds$ for bindings).

4 Bypassing Analysis

Through the bypassing analysis we want to detect those situations in which a variable is used exactly once as input channel and once as output channel (from the parent's point of view) and is not used in any other expression. This means that, in a process abstraction, those channels specified as child's inports are going to be registered as parent's outputs. This is quite similar to a usage analysis [Ses91], [LGH+92].

[1] Core is a minimal functional language to which Haskell is translated in the GHC's first steps

$$
\begin{array}{lll}
 & program \rightarrow binds & \\
 & binds \rightarrow bind & \\
 & \quad \mid \ \textbf{rec}\ bind_1; \ldots; bind_n & \\
* & \quad \mid \ \textbf{recpar}\ bind'_1; \ldots; bind'_n\ [\textbf{bypass}\ channels] & \\
 & bind \rightarrow var = exp & \\
 & bind' \rightarrow var = exp & \\
* & \quad \mid \ channels = var\ \#\#\ channels & \\
* & channels \rightarrow \{var_1, \ldots, var_n\} & \\
 & exp \rightarrow exp\ atom & \\
 & \quad \mid \ \lambda\ var \rightarrow exp & \\
 & \quad \mid \ \textbf{case}\ exp\ \textbf{of}\ alts & \\
 & \quad \mid \ \textbf{let}\ binds\ \textbf{in}\ exp & \\
 & \quad \mid \ C\ var_1 \ldots var_n & \\
 & \quad \mid \ prim\ var_1 \ldots var_n & \\
 & \quad \mid \ atom & \\
* & \quad \mid \ \textbf{process}\ channels \rightarrow body\ [\textbf{bypass}\ channels] & \\
* & body \rightarrow [\textbf{let}\ binds\ \textbf{in}]\ channels & \\
 & alts \rightarrow Calt_1; \ldots; C_a lt_n; Default & \\
 & \quad \mid \ Lalt_1; \ldots; Lalt_n; Default & \\
 & Calt \rightarrow C\ var_1 \ldots var_m \rightarrow exp & \\
 & Lalt \rightarrow Literal \rightarrow exp & \\
 & Default \rightarrow NoDefault & \\
 & \quad \mid \ var \rightarrow exp &
\end{array}
$$

Fig. 3. CoreEden and annotated CoreEden

We use an abstract bypassing domain $B^{\#}$, shown in Figure 4, where $i1o1$ represents that the channel is used only once as input and only once as output. In such a case the channel will be bypassed. The value N represents the fact that the channel's value is used too many times, for example twice as input or as a free variable.

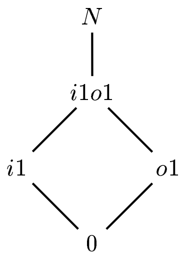
In some analysis functions we will use bypassing environments ρ. These are functions from program variables to bypassing values ($Env = Var \rightarrow B^{\#}$). In Figures 6 and 7 we consider that those variables not appearing in an environment ρ have a 0 bypassing value, and we denote the set of variables with a bypassing value different from 0 as $dom\ \rho$. We also define a $+$ operation over bypassing values, shown in Figure 4, which extends easily to environments. We also need a $+$ operation over sets of environments due to the fact that a variable may appear more than once as input channel and then bypassing is not possible.

The analysis uses three main functions, see Figures 6 and 7, to analise expressions (A_{exp}), bindings (A_{binds}) and process bodies ($A_{bindsbody}$). The other ones are auxiliar functions. We use v to denote variables, e to denote expressions, cs to denote channels and a to denote atoms. The rest of the notation is selfexplanatory. We also use two functions, $\Cap$ and $\curlywedge$, defined as follows: $cs \Cap ds = \{x \mid x \in cs \wedge \exists! y \in ds.y = x\}$ and $(f \curlywedge g)\ x = (f\ x) \wedge (g\ x)$. The first one is used to detect bypassing between ancestors.

The function A_{exp} receives a CoreEden expression e and decorates it with bypassing information, which in the end is attached to process abstraction expressions and **recpar** bindings in e. Such information is local to the process abstraction, so we analyse its body with a bypassing environment carrying all the information about input and output channels. In a **let** *binds* **in** e expression we analyse e and then the bindings, taking into account that the free variables in e cannot be bypassed (so we set all the free variables to N in the environment used to analise the bindings). In a **case** expression we decorate the discriminant and each of the expressions in the right hand sides of the alternatives, as we can have process abstractions to be decorated. For lack of space function A_{alts} is not detailed in Figure 7.

The functions A_{binds} and $A_{bindsbody}$ call the auxiliar function $A_{propagate}$ with different arguments. This function receives an environment and builds from it a new environment adding the first one to the combination (achieved by $A_{listbinds}$) of the informations obtained from each individual binding (given by $A_{onebind}$). In the first case it receives an environment in which all the free variables of the **let** main expression have N as bypassing value and the resulting environment is discarded. In the second case, it receives an environment in which the process abstraction channels have their corresponding $o1$ or $i1$ bypassing value, and the resulting environment is lately used to annotate the process abstraction.

Note that in a process abstraction, those channels involved in a bypassing between siblings appear in the annotations of the bindings, those involved in a bypassing between ancestors appear in the annotations of the **process** expression and those involved in a bypassing between generations appear both in the bindings and in the **process** expression.



$$
\begin{array}{l}
N + x = N \\
0 + x = x \\
i1o1 + x = N \ (x \neq 0) \\
i1 + o1 = i1o1 \\
i1 + i1 = N \\
o1 + o1 = N
\end{array}
$$

Fig. 4. Bypassing domain

We need a function *freevars* to obtain the free variables of an expression in two places: in a binding of the form $var = e$ and in a **let** *binds in* e expression, those variables free in e cannot be bypassed, so they must be set to a bypassing value of N. The definition of *freevars* is the usual one extended as shown in Figure 5 for the new constructions.

$$
\begin{array}{l}
\mathit{freevars}\ (\mathbf{process}\ \mathit{channels} \rightarrow \mathit{body}) = \mathit{freevars}\ \mathit{body} - \mathit{channels} \\
\mathit{freevars}\ (\mathit{channels}) = \mathit{channels} \\
\mathit{freevars}\ (\mathbf{let\ rec}\ x_1 = r1 \ldots x_n = r_n\ \mathbf{in}\ \mathit{channels}) = \\
\quad \bigcup_{i=1}^{n} \mathit{freevars}\ r_i \cup \mathit{channels} - \{x_1, \ldots, x_n\} \\
\mathit{freevars}\ (\mathbf{let\ recpar}\ \{x_i = r_1\}_{i=1}^{n}\ \{os_j = p_j\ \#\#\ is'_j\}_{j=1}^{m}\ \mathbf{in}\ \mathit{channels}) = \\
\quad \bigcup_{i=1}^{n} \mathit{freevars}\ r_i \cup \bigcup_{j=1}^{m} is_j \cup \bigcup_{j=1}^{m} p_j \cup \mathit{channels} - \bigcup_{i=1}^{n}\{x_i\} - \bigcup_{j=1}^{m} os_j \\
\mathit{freevars}\ (\mathbf{let\ recpar}\ \{x_i = r_1\}_{i=1}^{n}\ \{os_j = p_j\ \#\#\ is'_j\}_{j=1}^{m}\ \mathbf{in}\ \mathit{expr}) = \\
\quad \bigcup_{i=1}^{n} \mathit{freevars}\ r_i \cup \bigcup_{j=1}^{m} is_j \cup \bigcup_{j=1}^{m} p_j \cup \mathit{freevars}\ \mathit{expr} - \bigcup_{i=1}^{n}\{x_i\} - \bigcup_{j=1}^{m} os_j
\end{array}
$$

Fig. 5. Free variables function

Applying this algorithm to the (translated to CoreEden) example of Section 2, we would obtain the following annotations for the binding of *pipe*:

$$
\begin{array}{l}
\mathit{pipe} = \lambda\ \mathit{ps} \rightarrow \mathbf{case}\ \mathit{ps}\ \mathbf{of} \\
\qquad [p] \rightarrow p \\
\qquad p : \mathit{pp} \rightarrow \mathbf{process}\ z \rightarrow \mathbf{let\ recpar}\ \mathit{inter} = p\ \#\#\ z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad p' = \mathit{pipe}\ \mathit{pp} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad y = p'\ \#\#\ \mathit{inter} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{bypass}\ z, y, \mathit{inter} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ y\ \mathbf{bypass}\ z, y
\end{array}
$$

5 The bypassing protocol

The compile time analysis has delivered the information of the local forwards. In what follows, we will call *forward chain* to a connection from a real ouport (also called the *producer*) to a real inport (also called the *consumer*) through a sequence of intermediate local forwards. At process creation time, we must introduce these final ports to each other. Before explaining the process creation protocol with bypassing, the process creation protocol without it is presented.

$$
\begin{array}{l}
A_{exp} :: Exp \rightarrow DExp \\
A_{exp}\ (e\ a) = (A_{exp}\ e)\ a \\
A_{exp}\ (\lambda\ v \rightarrow e) = \lambda\ v \rightarrow A_{exp}\ e \\
A_{exp}\ (\mathbf{case}\ e\ \mathbf{of}\ alts) = \mathbf{case}\ e'\ \mathbf{of}\ alts' \\
\qquad \texttt{where}\ e' = A_{exp}\ e \\
\qquad\qquad\quad alts' = A_{alts}\ alts \\
A_{exp}\ (C\ v_1 \ldots v_n) = C\ v_1 \ldots v_n \\
A_{exp}\ (prim\ v_1 \ldots v_n) = prim\ v_1 \ldots v_n \\
A_{exp}\ (\mathbf{let}\ binds\ \mathbf{in}\ e) = \mathbf{let}\ binds'\ \mathbf{in}\ e' \\
\qquad \texttt{where}\ e' = A_{exp}\ e \\
\qquad\qquad\quad binds' = A_{binds}\ binds\ (freevars\ e) \\
A_{exp}\ (\mathbf{process}\ cs_1 \rightarrow cs_2) = \\
\qquad \mathbf{process}\ cs_1 \rightarrow cs_2\ byp \\
\qquad \texttt{where}\ cs = cs_1 \mathbin{\dot{\cap}} cs_2 \\
\qquad\qquad\quad byp = \texttt{if}\ cs = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ cs \\
A_{exp}\ (\mathbf{process}\ cs_1 \rightarrow \mathbf{let}\ binds\ \mathbf{in}\ cs_2) = \\
\qquad \mathbf{process}\ cs_1 \rightarrow \mathbf{let}\ binds'\ \mathbf{in}\ cs_2\ byp \\
\qquad \texttt{where}\ \rho_0 = \{c \mapsto o1 \mid c \leftarrow cs_1\} + \mathop{\Large +}\{\{d \mapsto i1\} \mid d \leftarrow cs_2\} \\
\qquad\qquad (binds', \rho') = A_{bindsbody}\ binds\ \rho_0\ (cs_1 \mathbin{\dot{\cap}} cs_2) \\
\qquad\qquad b = filter\ (== i1o1)\ (map\ \rho'\ (cs_1 \cup cs_2)) \\
\qquad\qquad byp = \texttt{if}\ b = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ b
\end{array}
$$

Fig. 6. The analysis function for expressions

5.1 The protocol without bypassing

The original communication protocol has been described in [KOMP98]. Here, we only present the process creation part, which uses the following messages:

- *CREATE-PROCESS(pabs,*ins_p*,*$outs_p$*)*: initiates the creation of a child process in a PE by using the process abstraction *pabs* and the parent's inports and outports, respectively ins_p and $outs_p$, which will be connected to the child.
- *ACK(*$outs_c \rightarrow ins_p, outs_p \rightarrow ins_c$*)*: acknowledges the creation of the child process to the parent, including the connections between parent and child.

While the child immediately starts sending values to the parent, the parent must wait for the *ACK* message in order to know to which child ports the values should be sent. Values are sent by using a *SENDVAL(in,value)* message. Figure 8 depicts the messages during the instantiation of a two stages pipeline from the example program in Section 2.

5.2 New RTS components: handles and forward table

To represent each local forward we use a unique identifier within a PE, called *forward handle*:

```
type HForward = (PE, Int, Level)
```

The third component (not really part of the identifier) is the level of the process in the process creation tree to which the handle belongs. It is an additional information whose utility will be explained in Section 5.3. Handles can take the place of inports and outports in the messages of the bypassing protocol.

To store intermediate connection information we add to the RTS a new runtime table:

```
type ForwardTable = HForward -> (Outport, Inport)
type Inport, Outport = GlobalPort | HForward
type GlobalPort = (PE, Int)
```

Additionally, we need to keep the process level in the state of the process to be able to create new forwards with the correct level.

$$
\begin{array}{l}
A_{binds} :: Binds \to Free \to DBinds \\
A_{binds}\ binds\ free = fst\ (A_{propagate}\ binds\ \{x \mapsto N \mid x \leftarrow free\}\ \emptyset) \\
\\
A_{bindsbody} :: Binds \to Env \to (DBinds, Env) \\
A_{bindsbody}\ binds\ \rho\ cs = A_{propagate}\ binds\ \rho\ cs \\
\\
A_{propagate} :: Binds \to Env \to Channels \to (DBinds, Env) \\
A_{propagate}\ (v = e)\ \rho\ cs = (bind', \rho + \rho') \\
\qquad \mathtt{where}\ (bind', \rho') = A_{onebind}\ (v = e) \\
A_{propagate}\ (\mathbf{rec}\ binds)\ \rho\ cs = (\mathbf{rec}\ binds', \rho + \rho') \\
\qquad \mathtt{where}\ (binds', \rho') = A_{listbinds}\ binds \\
A_{propagate}\ (\mathbf{recpar}\ binds)\ \rho\ cs = \\
\qquad (\mathbf{recpar}\ binds'\ byp, \rho'') \\
\qquad \mathtt{where}\ (binds', \rho') = A_{listbinds}\ binds \\
\qquad\qquad\quad \rho'' = \rho + \rho' \\
\qquad\qquad\quad b = filter\ ((== i1o1 \cdot get) \wedge (not \cdot in\ cs))\ (dom\ \rho'') \\
\qquad\qquad\quad byp = \mathtt{if}\ b = \emptyset\ \mathtt{then}\ \epsilon\ \mathtt{else}\ \mathbf{bypass}\ b \\
\\
A_{listbinds} :: [Bind] \to ([DBind], Env) \\
A_{listbinds}\ (bind : binds) = (bind' : binds', \rho' + \rho'') \\
\qquad \mathtt{where}\ (bind', \rho') = A_{onebind}\ bind \\
\qquad\qquad\quad (binds', \rho'') = A_{listbinds}\ binds \\
A_{listbinds}\ [] = ([], \emptyset) \\
\\
A_{onebind} :: Bind \to (DBind, Env) \\
A_{onebind}\ (v = e) = \\
\qquad (v = e', \{v \mapsto N\} + \{x \mapsto N \mid x \leftarrow freevars\ e\}) \\
\qquad \mathtt{where}\ e' = A_{exp}\ e \\
A_{onebind}\ (cs_1 = p\ \#\#\ cs_2) = (cs_1 = p\ \#\#\ cs_2, \rho') \\
\quad \mathtt{where}\ \rho' = \{c \mapsto o1 \mid c \leftarrow cs_1\} + \textstyle\sum\{\{d \mapsto i1\} \mid d \leftarrow cs_2\} + \{p \mapsto N\}
\end{array}
$$

Fig. 7. The analysis functions for bindings

5.3 The revised protocol

The scheme of the protocol is the following:

phase 1: processes are created in a tree–like fashion. In the *CREATE-PROCESS* messages, all available connection information, either real ports or descendant forwards, is propagated downwards through the inter–generational forwards.

phase 2: once the final ports of a chain are reached, this information is reported upwards in the tree.

phase 3: the consumer and producer of the chain are introduced to each other.

The messages In contrast to what happened in the old *CREATE-PROCESS* message, the ports now need not to belong to the parent itself and handles are also allowed. Apart from the *CREATE-PROCESS* message, there are three new bypassing messages:

- *CONSUMER(out* $\to$ *in)*: tells the producer which is the real consumer of the chain. Always, *in* is a real inport, whereas *out* may be an outport or a handle.
- *PRODUCER(out* $\to$ *in)*: tells the consumer which is the real producer of the chain. Always, *out* is a real outport, whereas *in* may be an inport or a handle.
- *LOOP(out* $\to$ *in)*: in this case, both *out* and *in* are descendant forwards. It is used in special cases in which an ancestor forward is involved in the chain. This is explained below.

The old *ACK* message is replaced by the new bypassing messages, so it is logically not needed anymore.

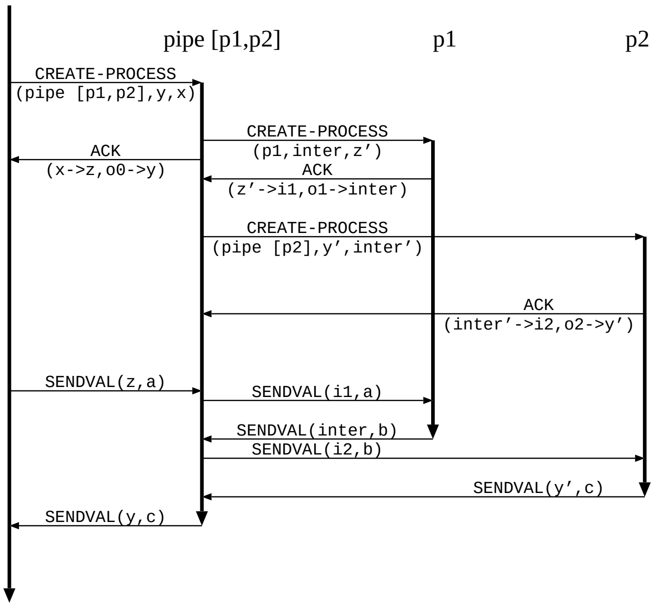


Fig. 8. Protocol without bypassing for a two stages pipeline

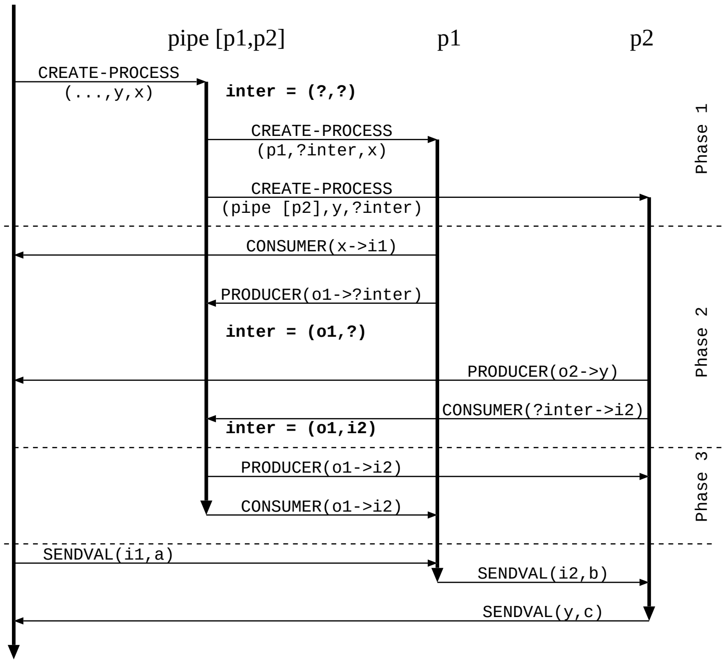


Fig. 9. Protocol with bypassing for a two stages pipeline

Phase 1: modified process creation In the presence of a descendant forward, some inports and/or outports will not be created. In the *CREATE-PROCESS* message, handles replace such absent ports. This happens, for example, in the second and third *CREATE-PROCESS* messages of Figure 9, where we denote handles by prefixing their names with a ? symbol.

If some connection information is already known, it is sent instead of the local handle. This is the case with inter-generational forwards. In Figure 9, the real outport `x` is propagated downwards through the descending forward of `pipe [p1,p2]`. Ancestor forwards are not involved in the process instantiation on the parent side.

After all processes have been created, we have the following situation: all inter-generational forwards have been removed. Each process at the end of a chain has received as opposite end either a descendant forward or a real port of an ancestor process.

Phase 2: sending bypassing messages upwards When a child process is the real producer/consumer of a chain, it must inform to the other side of the chain. This is done by a *CONSUMER/PRODUCER* message. If an outport already knows its consumer, it can directly start sending data. In Figure 9 these situations are reflected by the four upwards messages.

In this phase we also take care of the ancestor forwards. When at least one of the ends is a real port, we pass the information to the other end by using again a *CONSUMER/PRODUCER* message. A special case arises when both ends are connected to descendant forwards. This is separately explained below.

Phase 3: the final connection messages When a descendant forward has received the messages from the real consumer and the real producer of the chain, it introduces them to each other by also sending a *CONSUMER* and a *PRODUCER* message. In Figure 9 these situation is reflected by the next two downwards messages.

The special case: loop A *loop* consists of an ancestor forward connecting two descendant forwards as illustrated in Figure 10, where `c` connects `a` and `d`. The descending forward `b` has already been deleted when `a` was passed on to the grandchild. For this special case a

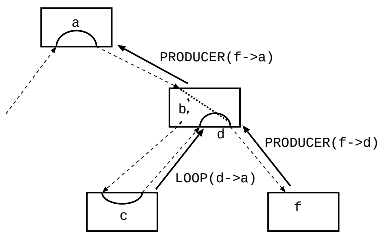


Fig. 10. Special case: Loop

LOOP message is sent to the descendant forward in the lower process. The upper handle will eventually be informed by the lower one. This is the reason why we need the level information in handles. After receiving the *LOOP* message, the reaction is different according to the class of receiver's forward. If it was directed to

- a descendant forward, this is transformed into an inter-generational forward (this is the only case in which an inter-generational forward may exist in the second phase);
- an inter-generational forward, this is transformed into an ancestor forward, which immediately sends another *LOOP* message upwards.

5.4 Overall communication costs

We said before that the old *ACK* message was not needed anymore, but in the real implementation it is still used to collect several bypassing messages into one message to the parent. Then, if the program contains no forwards, the messages in the old and in the revised protocol are identical, i.e. the creation of tree topologies costs exactly the same.

For each descendant forward in the chain, two messages are needed in the second phase and two additional messages in the final connection phase. In chains not containing descendant forwards, only one message is needed from the lower to the upper end.

Some optimizations, orthogonal to the protocol, are possible to reduce the number of bypassing messages. For instance, the RTS will not actually send messages that are local to a PE. It will also pack in a single physical message several logical messages addressed to the same PE. We have not yet measured the savings of these optimizations.

6 Conclusions

The expected savings of bypassing come from the fact that the number of data messages is usually much greater than the number of protocol messages needed to create the direct connection. This is because, when programming in Eden, many of the channels are lists whose transmission implies as many data messages as list elements. Even in the case of one-value channels, the number of data messages may be greater than one depending on the size of the transmitted value.

For a forward chain of length $n, n \geq 2$, a total of $n-1$ messages are saved for every data message sent through the direct connection. In the other hand, the number of additional messages of the new protocol, with respect to the old one, varies from 0 to an unbounded number depending on the chain complexity. For a typical chain consisting of a desdendant forward, and at least two inter-generational forwards, this number is 4. In this case, the length of the chain is greater than or equal to 4. We must add one more message, and a length of two to the chain, for every ancestor forward. The savings must also take into account the overheads of the threads not created in all the intermediate processors.

In conclusion, we can say that, for streams and big values, there are always substantial gains. Even in the worst case —one-value channel fitting in one data message— the savings are clear when the length of the forward chain is greater than 4. As it is common practise to define process topologies by using 'intermediate processes', mostly in combination with higher order functions [GPP96], these cases are frequent enough so that the optimization is worthwhile.

The current state of the implementatios is as follows: the RTS has already been modified for the new protocol as it is under debugging. The compile time analysis and the many transformations involved to produce a 'good' CoreEden are still being implemented.

References

[BKL98] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An implementation point of view. In PLILP'98. Springer LNCS 1490, pages 318–334, 1998.

[BLOMP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

[GPP96] L. A. Galán, C. Pareja, and R. Peña. Functional Skeletons Generate Process Topologies in Eden. In *Int. Symp. on Programming Languages, Implementations, Logics and Programs PLILP'96. Aachen, Germany. LNCS Springer-Verlag no. 1140*, pages 289–303, 1996.

[JHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[KOMP98] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. LNCS 1595, Springer-Verlag*, pages 1–16, 1998.

[LGH+92] J. Launchbury, A. Gill, R. J. M Hughes, S. Marlow, S. L. Peyton Jones, and P. L. Wadler. *Avoiding Unnecessary Updates*. Springer-Verlag Workshops in Computing, New York, NY, 1992.

[Ses91] P. Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, October 1991.