Porting the Eden System to GHC 5.00 *

Jost Berthold, Rita Loogen, Steffen Priebe, and Nils Weskamp

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,loogen,priebe,weskamp}@mathematik.uni-marburg.de

Abstract. This paper presents selected implementation aspects of the parallel functional language Eden, based on the Glasgow Haskell Compiler. Written in the context of porting Eden to a new GHC-version, it focusses on the implementation principle by different layers, on primitive operations and their support within the runtime system. It also presents preliminary runtime results comparing the old and the new system.

1 Introduction

A growing number of applications demand a large amount of computing power. This calls for the use of parallel hardware and the development of software for these architectures. Parallel programming is however hard. The programmer usually has to care about process synchronization, load balancing, and other low-level details, which makes parallel software development complex and expensive.

Our approach, Eden [5], has the aim to use the advantages of functional programming to simplify the development of parallel software, while giving the programmer explicit control over the parallel behaviour of a program. The current implementation of the Eden system is based on the Glasgow Haskell Compiler (GHC) [13]. The Eden system is composed of the following layers:

Eden Programs
(Skeleton) Libraries
Eden Module
Primitive Operations
Parallel Runtime System

Eden programmers will typically develop parallel programs using the Eden language constructs briefly described in Section 2, together with parallel skeletons provided in special libraries [10]. Every Eden program must import the Eden module, which contains Haskell definitions of Eden's language constructs as explained in Section 3. These Haskell definitions heavily use primitive operations which are functions implemented in C that can be accessed from Haskell. They implement the functionality needed by Eden's constructs. Interfacing Eden to Haskell via the Eden module means that the front-end of the GHC need not

* Work supported by the DAAD (Deutscher Akademischer Austauschdienst)

be modified to compile Eden programs. The extension of the GHC for Eden is mainly based on the implementation of appropriate new primitive operations, which is discussed in Section 4. The core of the Eden system is the parallel runtime system which has been developed by modifying the GUM runtime system [17] of Glasgow parallel Haskell (GpH) [16] for Eden [3, 8]. This will also be discussed in Section 4.

This paper has been written in the context of porting our Eden implementation from GHC version 3.02 to version 5.00. This was necessary because, from version 4.00 on, the GHC runtime system has been largely revised and improved [15]. In particular, the new storage manager supports a dynamic adaptation of the heap size and Concurrent Haskell is supported by default. The main task of the porting project was to implement the needed support for Eden into the new runtime system, having the old system with the Eden extensions as a reference. However, analyzing the old system revealed that some implemented solutions in the old system either could not be kept, or had to be considered as "second best", having the improved methods of the new runtime system at hand. So the project included a certain re-design of system details. Due to the fact that GHC in general is hardly documented on implementation level, we may note that it also became a trip into the gory details of some GHC components. Finding out how GHC behaves is always interesting, but a time-consuming job. In Section 5, we show the results of some runtime measurements and summarize our main experiences.

To follow GHC's line of development is essential for Eden's future portability. The interface between GHC and Eden can be minimized to only eight primitive operations, which provide the elementary functionality for Eden. We hope that this minimal interface will help us to keep up with the GHC development. The primitive operations are based on orthogonal extensions of the sequential runtime system using infrastructure from GpH. Our approach will be of interest for others who want to extend GHC or a comparable system as part of their own work.

2 Eden's Main Features

Eden [5] extends the lazy functional language Haskell [14] by syntactic constructs for *explicitly* defining processes. Eden's process model provides direct control over process granularity, data distribution and communication topology. The Eden syntax has been changed in comparison to previous papers to achieve a smoother integration into Haskell. The underlying semantics and concepts are the same as before.

Basic Constructs. A *process abstraction* expression `process (\x -> e)` of type `Process a b` defines the behaviour of a process having the parameter `x::a` as input and the expression `e::b` as output. The function

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
```

embeds functions of type `a -> b` into process abstractions of type `Process a b`[1]. The main difference between functions and process abstractions is that the latter, when instantiated, are executed in parallel. The context `Trans a`[2] ensures that functions for the transmission of values of type `a` are available.
A *process instantiation* uses the predefined infix operator

```
( # ) :: (Trans a, Trans b) => Process a b -> a->b
```

to provide a process abstraction with actual input parameters. The evaluation of an expression `(process (\ x -> e1)) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* will evaluate the application `(\ x -> e1) e2` and return the result via another implicitly generated channel. Note that communication is always invisible for the programmer.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access input which is not available yet are temporarily suspended. This is the only way in which Eden processes synchronize.

Example: The following program computes, for a given integer `n`, the sum of a list of so-called Euler values: `sum (map euler [n, n-1..1])`.
The function `euler :: Int -> Int` computes the number of integers that are relatively prime to a given integer.

```
sumEuler  :: Int -> Int -> Int
sumEuler c n = sum xs ‘using‘ spine
               where xs = [ (process (sum . map euler)) # chunk
                          | chunk <- splitAtN c [n, n-1..1] ]
```

The list `[n,n-1..1]` is split into chunks of size `c`. For each chunk, a process is created which computes the corresponding part of the total sum. The addendum `‘using‘ spine` is needed to produce early demand for the evaluation of the process instantiations. ◁
Non-strictness, implemented by using lazy evaluation of expressions, is a key point in our approach. Lazy evaluation is changed to eager evaluation in two cases: processes are eagerly instantiated, and instantiated processes produce their output even if it is not demanded. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation. In

[1] The previous syntax of a process abstraction was `process x -> e`. Handling `process` as a special function instead of as a new syntactic construct has many advantages. The process function simply transforms functions into process abstractions. Thus, a single definition now suffices to specify both a function and the corresponding process abstraction.
[2] The context was previously called `Transmissible`.

general, a process is implemented by several threads concurrently running in the same processor, so that different values can be produced independently. The concept of a virtually shared global graph is avoided, to save the administration costs while paying the price of possibly duplicating work. Each process evaluates its outputs autonomously.

Extensions. The base system described above has been extended in different ways to make programming in Eden more convenient and to improve the expressive power of the language. *Many-to-one communication* is an essential feature for some parallel applications, but it spoils the purity of functional languages, as it introduces non-determinism. In Eden, the predefined process abstraction

```
merge :: Trans a => Process [[a]] [a]
```

is used to instantiate a process which does a fair merging of input streams into a single (non-deterministic) output stream. The incoming values are passed to the output stream in the order in which they arrive.

An Eden process may also explicitly generate a new *dynamic input channel* and send a message containing the channel's name to another process. The receiving process may then either use the name to return some information to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Both possibilities exclude each other, and a runtime error will occur if not appropriately used.

Eden introduces a new unary type constructor `ChanName` for the names of dynamically created channels. Moreover, it also adds a new operator[3]

```
new :: Trans a => (ChanName a -> a -> b) -> b
```

Evaluating an expression `new (\ (ch_name, ch_vals) -> e)` has the effect that a new channel name `ch_name` is declared as reference to the new input channel `ch_vals`, which represents future input. The scope of both is the body expression `e` whose value is the result of the whole expression. The channel name should be sent to another process to establish the communication. A process receiving a channel name `ch_name`, and wanting to reply through it, uses the function[4]

```
parfill :: Trans a => ChanName a -> a -> b -> b
```

Evaluation of an expression `parfill ch_name e1 e2` means: Before `e2` is evaluated, a new concurrent thread for the evaluation of `e1` is generated, whose normal form result is transmitted via the dynamic channel. The result of the overall expression is `e2`, while the generation of the new thread and its communication through the dynamic channel is a side effect.

[3] In the previous syntax, a syntactic construct `new` has been used. As with process abstractions, we replace the old `new` construct by a special function `new`.

[4] The previous syntax used a mixfix operator `_ !* _ par _` which now has been renamed to `parfill`, as infix operators with three arguments cannot be defined in Haskell.

Skeletons are a special form of higher-order function with a parallel implementation that simplify the development of parallel programs. A good example is the well-known `map` function, which applies its argument function to each element of a given list. As each of these calculations is independent, the evaluation of each element of the result list can be done in parallel. The Eden libraries offer a number of parallel implementations for `map` (and other parallel skeletons) that can be substituted for a sequential `map` and lead to an instant parallelized version of the program.

3 The Eden Module

The Eden module which must be imported by every Eden program contains Haskell definitions for the Eden constructs. These definitions use primitive operations which provide the needed functionality. The interface of the Eden module is as follows

```
module Eden (
    Trans(...), Process, process, ( # ),
    ChanName, new, parfill, ...    )   where
```

Subsequently, we will focus on the module definitions for process abstraction and instantiation shown in Fig. 1, which are at the center of the whole system. Fortunately, process creation can be defined completely using the primitive operations and functions implementing dynamic channels plus one additional primitive operation `createProcess#`[5] for forking a process on a remote processor.

A process abstraction of type `Process a b` is implemented by a function that will be evaluated remotely when a corresponding child process is created and that takes two channel names as arguments. The first argument of type `ChanName b` is a channel for sending its output while the second argument of type `ChanName (ChanName a)` is a channel to pass the name(s) of its input channel(s) to the parent process. The number of channels that will be established between parent and child process does not matter in this context, because the operations on dynamic channels are overloaded. Tuple types `a` or `b` lead to the creation of a channel for each tuple component. The definition of `process` (see Fig. 1) shows that the remotely evaluated function, `f_remote`, creates its input channels via a function

```
createDC :: Trans a => a -> (ChanName a, a)
```

This function yields (a list of) channel names `inDCs` that will be returned to the parent process and (a handle to) the values `invals` that will be received via these channels. To ensure correct typing of the program, `createDC` is applied to its second output, but will not make use of its argument except for determining

[5] Note that primitive operations in GHC are distinguished from common functions by `#` as the last sign in their names.

```
data (Trans a, Trans b) =>
     Process a b = Proc (ChanName b -> ChanName (ChanName a) -> ())
process :: (Trans a, Trans b)
           => (a -> b) -> Process a b
process f = Proc f_remote
    where f_remote outDCs chanDC
           =  let (inDCs, invals) =  createDC invals
              in  writeDC chanDC inDCs ‘fork‘
                  (writeDCs outDCs (f invals))

( # ) :: (Trans a, Trans b) => Process a b -> a -> b
pabs # inps = case createProcess (-1#) pabs inps of Lift x -> x
data Lift a = Lift a

createProcess :: (Trans a, Trans b) =>
                 Int# -> Process a b -> a -> Lift b
createProcess on# (Proc f_remote) inps
        = let  (outDCs, outvals) = createDC outvals
               (chanDC, inDCs )  = createDC inDCs
               pinst = f_remote outDCs chanDC
          in outDCs ‘seq‘ inDCDC ‘seq‘
             case createProcess# on# pinst of
                 1# -> writeDCs inDCs inps  ‘fork‘ (Lift outvals)
                 _  -> error "process creation failed"
```

Fig. 1. Haskell definitions of Eden process abstraction and instantiation

the number of channels that have to be produced. The latter is done via an overloaded function `tupsize` that gives the number of components of a tuple. The function

```
writeDCs :: Trans a => ChanName a -> a -> ()
```

takes (a list of) channel names and a (tuple) expression[6] and creates a thread for each channel name which evaluates the corresponding tuple component to normal form and sends the result via the channel. The latter is encoded in the function `writeDC`.

In the function implementing a process abstraction, `writeDC(s)` is used twice: the dynamically created input channels of the child, `inDCs`, are sent to the parent process via the channel `chanDC` and the results of the process determined by evaluating the expression `(f invals)` are sent via the channels `outDCs`.

Process instantiation by the operator `( # )` defines the process creation on the parent side. To cope with lazy evaluation and to get the control back without waiting for the result of the child process, the process results are lifted to an

[6] The size of the tuple is identical to the number of channel names.

immediately available weak head normal form using the constructor `Lift`. Before returning the result the Lift is removed. The function `createProcess` takes the process abstraction and the input expression and yields the lifted process result. The placement parameter `on#` is an un-boxed integer (type `Int#`) which can be used to allocate newly created processes explicitly. The current system does not make use of this possibility. The channels[7] are handled using `createDC` and `writeDCs` in the same way as on the child side (see the process abstraction). The remote creation of the child process is performed by the primitive operation `createProcess#`.

A complete discussion of the Eden module would exceed the scope of this paper. The interested reader is referred to [7, 2] for more detailed descriptions. In summary, the Eden module gives a high-level definition of Eden's functionality by basing the implementation on eight primitive operations. The following section will explain the primitive operations themselves and their implementation in the parallel RTS, two subjects which cannot be cleanly separated.

4 Implementation of Primitive Operations

Just as in the previous section, we will only describe selected aspects of the underlying Eden support inside the new RTS, but we will mention every important primitive, passing from obvious issues to those involving more complexity hidden inside the runtime system.
Reflecting upon the necessary actions for a remote process to take place, it is necessary to

- create and connect communication channels in the proper way,
- explicitly request process instantiation (on another processor),
- pass input data to child processes,
- return (fully evaluated) results to parent processes,

and, last but not least,

- provide information about the system setup to the internal process-controlling functions in the Eden module and libraries.

As we have shown, the Eden functionality is driven essentially from language level by functions in the Eden module. Nevertheless, the underlying operations have to be performed directly in the runtime system of GHC, in the form of primitive operations, because either the needed information is hidden inside it, or the action mainly consists of side effects and cannot be specified in a functional manner.

[7] The prefixes in and out in channel names in Figure 1 reflect the point of view of a child process. Thus, in a process instanciation, the inputs `inps` for the child are written into the channels `inDCs`, which are outputs of the parent process.

4.1 Passing system information to language level

The only system information we want on language level is the number of available processors and the own processor ID, provided by `noPE#` and `selfPE#`. Both are implemented in a rather trivial manner: Using routines of a message passing system (PVM), the runtime system must, of course, contain variables which contain the desired information. These variables are just passed up to Haskell as integers.

Information provided by these primitives should be used only for process control. It is clear that, if the operations are misused, this introduces nondeterministic effects into the functional language. Defining a function whose result depends on the system setup is not intended and therefore explicitly prohibited. Eden's layer concept allows the development of independent Haskell libraries, an advantage we want to encourage by providing the easiest possible support.

4.2 The runtime tables: processes and communication ports

The GHC concurrency model relies on threads, which are internally represented by so-called *TSOs* (thread-state-objects). The basic idea of threads is that they perform autonomous evaluation of a reachable subgraph, always sharing the global graph heap with other threads.[8] In the RTS, an Eden process is modeled as a group of TSOs, which are working on the same process instantiation and on a single processor. The threads of a process in the RTS are connected using unique process identifiers (pids, unique per processor).

As described in [4] and [8], processes are closed entities that do not share any data among each other. Hence, processes only communicate through channels and use the pids to address them. The RTS equivalent to channels is a structure *Port*, which contains a specific Port number, the pid, and the processor number, thereby making port addresses unique. At module level, these port addresses (channels) are represented by the type

```
data ChanName' a = Chan Int# Int# Int#
type ChanName a = [ChanName' a]
```

`ChanName'` objects contain exactly the port identifier triple. Note that the type `ChanName a` used in the module is a list of port identifiers; one for each component of type `a` in case `a` is a tuple.

Two kinds of ports are distinguished: *Inports* are represented by the Haskell type `ChanName'`. They belong to a specific process, but are not linked to a special thread. An inport address (as a `ChanName'`) can be sent through a channel to allow higher-order process communication. In contrast, *outports* are related to TSOs, allowing the corresponding thread to be connected to a receiver and to send values. They could have the same representation as inports, but there is no point in representing them at language level. Both types of ports are administered by the runtime system using *runtime tables.*

[8] For further details on thread handling in GHC see [11]. Unlike GUM [17], threads are executed round robin in Eden.

The basic communication concept of the Eden RTS is to connect an inport of one process to exactly one outport of another to establish communication. Closing an inport (on garbage collection) implies that the connected outport (if any) sends unnecessary data and can therefore be closed. Closing an outport of a process implies stopping the connected TSO. If the last outport of a process is closed, it terminates by closing all its inports (which can lead to a termination cascade). Hence, runtime tables are involved in every operation on channels (creation, port connection, data transfer), including process creation/termination. This actually is the heart of the Eden RTS.

4.3 Channel handling in the runtime system

The essential functionality behind a channel is situated inside the runtime system: each channel inport is connected to a placeholder node (a so-called *QueueMe* closure), which stands for an expected result from another process and blocks any thread trying to evaluate it before arrival. Therefore, just creating a node of type `ChanName'` will not have any effect; it has to be connected in some way to the QueueMe, involving primitive heap operations to update it later. Objects of type `ChanName' a` and `ChanName a` are just handles to pass channels to other processes.

As a general rule, every primitive operation is wrapped in a function, thereby providing additional type information, triggering execution and limiting actual use to the desired degree. The function `createDC` relies on a primitive operation `createDC#`, which generates a tuple of QueueMe closures (the parameter indicating its arity) and a list (`ChanName`) of channels with inports connected to these Queue-Me closures inside the RTS. As explained in Section 3, the wrapper function `createDC` ensures correct typing of the created closures, which are generic in the runtime system.

```
createDC :: a -> (ChanName a, a)
createDC t = let (I# i#) = tupSize t in case createDC# i# of
                                          (# c,x #) -> (c,x)
createDC# :: Int# -> (# ChanName a,a #)
```
[9]

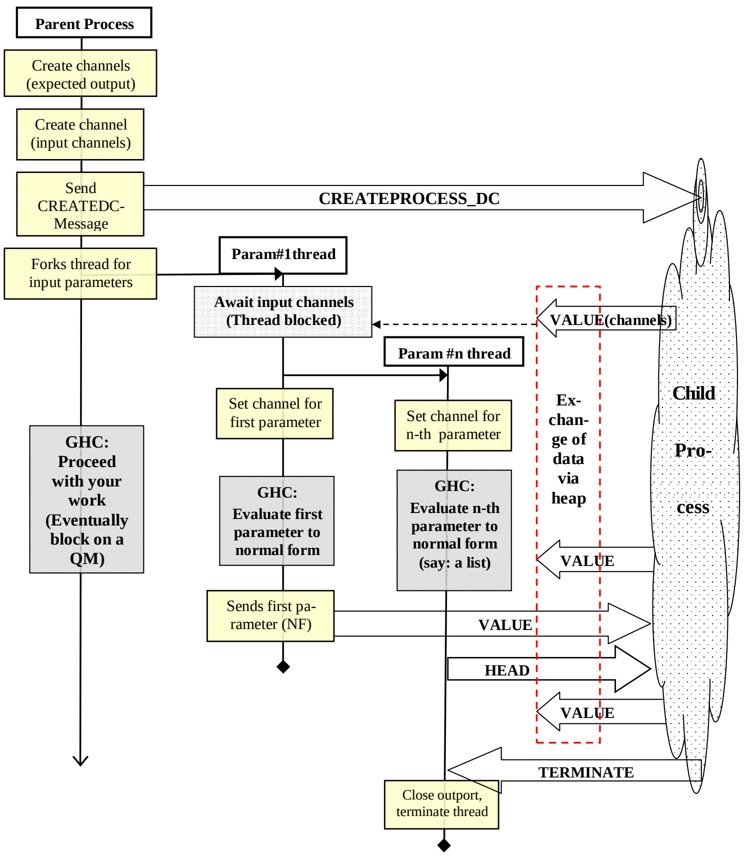
Once the inports are created, senders can be connected to them to send data input. Those senders will be threads in a remote process.

4.4 Data communication between processes

Processes in Eden have to exchange arguments and results, which is done by packing subgraphs[10] after their evaluation. Data transfer is performed by the primitive operations `sendVal#` and `sendHead#`, the latter sending one element of

[9] `(#..#)` indicates a so-called *un-boxed tuple*, a tuple which is handled directly by the registers of the abstract machine.

[10] This is a common issue to GpH and Eden. We modified the packing algorithm of GpH, eliminating all cases where shared data would be globalised and requested in a further transmission.



Parent Process
Create channels (expected output)
Create channel (input channels)
Send CREATEDC-Message
CREATEPROCESS_DC
Forks thread for input parameters
Param#1thread
Await input channels (Thread blocked)
VALUE(channels)
Param #n thread
Set channel for first parameter
Set channel for n-th parameter
Ex-change of data via heap
Child Pro-cess
GHC: Proceed with your work (Eventually block on a QM)
GHC: Evaluate first parameter to normal form
GHC: Evaluate n-th parameter to normal form (say: a list)
VALUE
Sends first parameter (NF)
VALUE
HEAD
VALUE
TERMINATE
Close outport, terminate thread

Fig. 2. Parent process on process instantiation

a list instead of a complete result. Both operations are used by an overloaded wrapper function `sendChan` to distinguish lists from simple values. This latter function is called by `writeDC`.

```
writeDC chan a = setChan chan ‘seq‘ sendChan a
```

Note that, before any evaluation takes place, the outport of the TSO must be connected to an inport of another process by calling a primitive `setChan#`. This connection prior to evaluating and sending values guarantees that the receiver of data messages is always defined when `sendVal#` or `sendHead#` is called. Sending without a valid receiver can never happen and would be discarded by the RTS.

Primitive sending operations, as seen from the implementation, can send any argument in the current evaluation state. The desired functionality is to send only normal form data and processes (`Trans`). This is guaranteed by the wrapper function `sendChan` in the module which always evaluates data to normal form before sending.

4.5 Handling channels for process instantiation

The most complex operation of the runtime system is process creation, driven by the primitive `createProcess#` and the semantics of the associated message `createProcessDC`. Creating a remote process involves packing and transferring a subgraph and creating an initial TSO on a remote processor when the message is received.

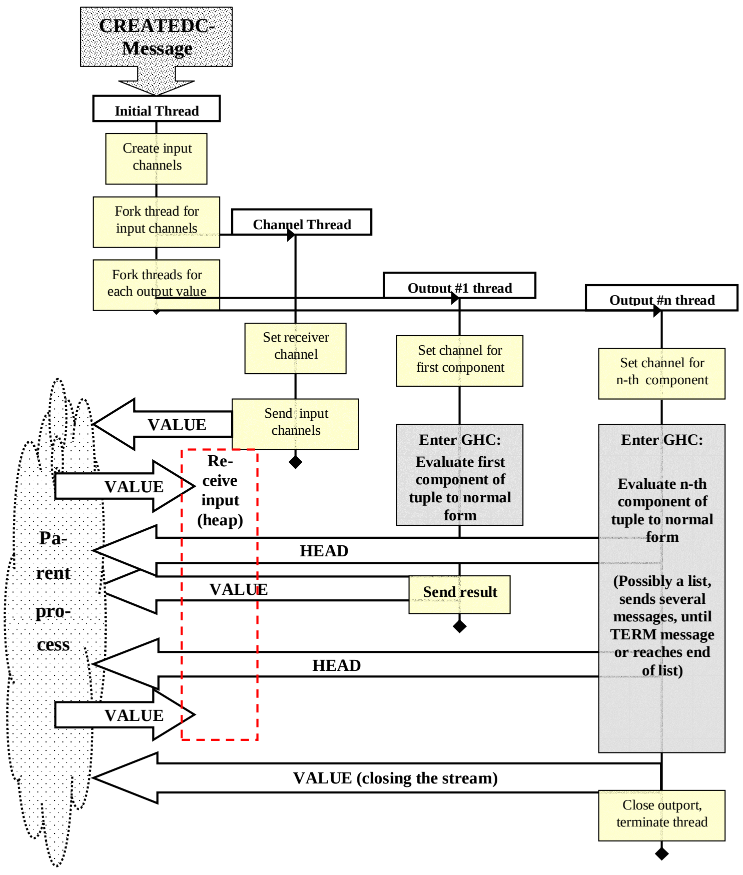

CREATEDC-Message
Initial Thread
Create input channels
Fork thread for input channels
Fork threads for each output value
Channel Thread
Set receiver channel
Send input channels
Output #1 thread
Set channel for first component
Enter GHC: Evaluate first component of tuple to normal form
Send result
Output #n thread
Set channel for n-th component
Enter GHC: Evaluate n-th component of tuple to normal form
(Possibly a list, sends several messages, until TERM message or reaches end of list)
Close outport, terminate thread
Re-ceive input (heap)
Pa-rent pro-cess
VALUE
VALUE
HEAD
VALUE
HEAD
VALUE
VALUE (closing the stream)

Fig. 3. Child process on process instantiation

The whole sequence of actions is shown in Figures 2 and 3. As explained in Section 3, process creation is preceded by the creation of new channels (one for every result) plus one additional port to receive channels for input parameters upon creation. The primitive `createProcess#` will then send a message `createProcessDC` to a remote processor, which contains these channels and the process abstraction (an unevaluated `Proc f_remote` packed as a subgraph, see Figure 1).

The remote processor receives this message, unpacks the graph and starts a new process by creating an initial thread. As the first thread in a process, this thread actually plays the role of a *process driver*. It creates channels for input, communicates them to the parent process and forks other TSOs for evaluation of the included function (which may need their arguments and will block on QueueMe-Closures). The sequence diagrams (Figures 2 and 3) illustrate in detail how a new process is created, sends and receives data and terminates. Individual threads can be terminated on request if their output turns out to be unnecessary. The diagrams are rather detailed concerning RTS actions, which would lead us too far in the textual description.

5 Results

5.1 Measurements with the new Eden system

The main reason for the presented port project has been to catch up with the GHC development and to allow the use of new GHC features in the future development of Eden. The port project revealed some weak points in the former Eden implementation which could be eliminated at this opportunity, as we now benefit from certain APIs and better structuring of the code (see 5.2). On the other hand, better structuring of the code does not imply better performance.

We carried out some simple performance comparisons to quantify the relationship between the old and the new system. Comparing the new implementation as a whole to the old one is in a sense inexact, since various other changes besides the Eden part must be taken into account, nor is the new runtime system optimized in any way. We present measurements of a raytracer program which had already been used for comparisons between GpH and Eden [9]. We repeated these previous measurements with a scene of 640 spheres and the direct mapping skeleton. Both programs use the same code (except for the new syntax), and up to 16 processors.

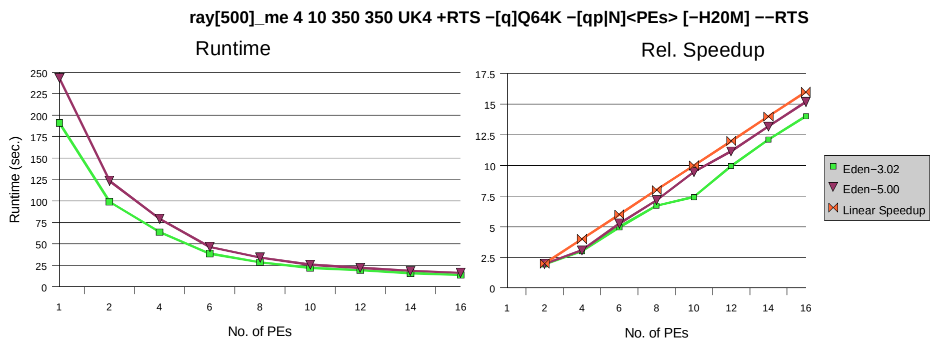


Fig. 4. Runtime and speed-up comparison (parallel Raytracer)

The sequential system is almost as fast as the old one (GHC 3.02: 186.2 s, GHC 5.00: 189.27 s). As shown in Fig. 4, the new Eden RTS is slightly slower than the old one. The runtime increased by a factor around 1.2, especially on few processors.

The overall ideas of implementation have not been changed in the new RTS, modifications only concern details. A possible reason could be that the new system is not optimized to avoid local data transfer via the message passing system. Further investigations will be necessary to determine other possible bottlenecks in the new RTS.

It is clear that we still need more experience and will soon present consolidated findings on how the new Eden RTS behaves, including further optimizations as e.g. enabling the generational garbage collection[11] and optimizing process communication.

5.2 Experiences with the GHC

Starting to work on a highly complex system software as the GHC will always encounter difficulties. It appears to us that sharing our experiences will supply an important feedback to other people involved in GHC, touching some tender points in the development.

Problems during the implementation. Implementing the Eden support in the RTS cannot be considered as a really complicated task, as long as the needed information was available. Having the old system at hand, we did not have to develop basic ideas of implementation. However, many things had to be picked up directly from the C-code. Exploring the details in the code sometimes led us to a different approach, having the possibilities of the new RTS. In particular, we do not use the so-called GpH-*spark-pool* any more, but create TSOs directly with the `fork`-operation and the RTS-API. We also entirely respecified the message protocol, which is now uniform with the GUM protocol.

The most critical part in the Eden system is the creation of new QueueMe-closures and channels via native C code; as this is the main difference from GpH/GUM functionality. Creating closures directly in C code implies knowledge of the appropriate info-pointers, taking care of the heap consumption and returning the created closures. The new runtime system provides suitable API-like functions and generic closure data structures which facilitate these issues.

The closure creation is closely related to the implementation of primitive operations in general and revealed that debugging the RTS evaluator with a common C debugger, gdb, is a rather difficult job, considering the machine-dependent optimizations and assembler mangling which strongly confuses gdb.

Another problem, which is still more general, is to understand the modularity of GHC and its RTS. Not only in the case of certain conventions in implementing

[11] GpH does not use generational GC yet. In the Eden implementation, generational GC would lead to later process termination if we did not provide a different solution to it.

primitive operations, but generally during our work on the runtime system, our main problem has always been to understand the functionality of GHC and to discern where we had to change it to achieve the desired behaviour. Compared to the concrete issues we had to realize, this task took us almost twice the time than the actual implementation. Most publications on GHC and its runtime system do not contain details an implementor needs to know, since they describe things on a very high level of abstraction.

Descriptions of the RTS implementation. In fact, there *are* some documents which describe the real implementation of the GHC RTS at a level which is useful for implementors:

1. **"The Stg Runtime System"** [11], the overall description of the runtime system, which is part of the GHC source distribution.
   Regardless of its intermediate state, this text is a very useful source of information about the RTS. Unfortunately, many passages are just stubs ("ToDo") and do not seem to be updated in the near future. In fact, it has not substantially changed since 1999.
2. **"Stg Survival sheet"** [1], mainly a developer's reference to all important parts of the GUM system and the RTS in general.
   This document has considerably grown since our first contact with the GHC runtime system. In its actual state, we consider it as the best reference to the GUM implementation, and very good for a quick start when hacking in the parallel RTS.
3. The **abstract machine description** in [12] and [15] involves some technical details of implementation.
4. The **GHC Commentary** [6] , which mainly covers the frontend, also contains some information about the RTS and the compiler as a whole.
5. **The code itself** contains many useful comments. The only problem is to find the file containing the comment one is looking for.

In the recent past, more attention has been paid to documentation issues, considering e.g. the low level documentation [1] for GUM and the fact that the commentary [6] has recently joined the repository. This definitely is a step in the right direction. Our own parts have been continuously documented in a working document, which will now be translated into English and brought into its final form [2].

6 Conclusions and Future Work

The aspects discussed in this paper focus on the division of the Eden implementation into different layers. After all, the Eden concept of lifting explicit control to functional level shows good parallel performance results [9]. It also makes development of extensions much easier once the RTS support is implemented.

Lifting aspects of the runtime system to functional level always implies a decision about what *must* remain hidden and what can be useful for library developers or Eden programmers. We have shown that the wrapper functions we use in the module are necessary in regard to typing issues, but also needed to restrict the implemented functionality to the desired degree. Implementing such restrictions directly in the RTS would be an alternative, but at a very high price in complexity and maintenance.

Seen from the high level perspective, the implementation of Eden relies on a few primitive operations and reuses much infrastructure of GpH, reducing it when necessary. Those changes have been very small, but hidden deeply in the RTS. Our documentation [2] describes what we actually implemented for Eden in the RTS as well as the problems we met, thereby facilitating comprehension of the low-level details.

The fact that there is only moderate documentation on the GHC implementation in general is an apparent shortcoming. The cited descriptions of the RTS, as well as the code itself, clearly show an underlying modular design, but this design is sometimes inconsistent in the encoding. No document fills the gap between the description of the RTS functionality in [11] and the code itself. It is clear that such a documentation would be of great help for new developers working on GHC. It can also be a certain guideline to the development altogether.

In the near future, we especially plan to optimize the communication. Further measurements and analysis will hopefully give us some hints at weak points in our implementation.

7 Acknowledgements

The Eden porting project would not have survived in the GHC jungle without the encouraging and supportive help of Hans-Wolfgang Loidl who never lost patience in explaining the details and mysteries of GHC to us. In addition Simon Marlow showed us the way out of some dead alleys. Fernando Rubio provided helpful comments on a draft version of this paper. We gratefully acknowledge their support and contributions to the work reported in this paper.

References

1. A poor wee Soul[sic]. The stg survival sheet. Unpublished, available at http://www.cee.hw.ac.uk/~dsg/gph/docs/StgSurvival.ps.gz.
2. J. Berthold and N. Weskamp. The Eden Porting Project - Porting the Eden Runtime-System from GHC 2.10 to GHC 5.00.2. draft report, Philipps-University Marburg, 2002. available at http://www.mathematik.uni-marburg.de/inf/eden/.
3. S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98 – Progr. Lang.: Impl., Logics and Programs*, LNCS 1490, pages 318–334. Springer, 1998.
4. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. DREAM - the DistRibuted Eden Abstract Machine. In *IFL '97 — Intl. Workshop on the Impl. of Funct. Lang.*, LNCS 1467, pages 250–269. Springer, 1997.

5. S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Progr. Models*, pages 120–124. IEEE Comp. Science Press, 1997.
6. M. Chakravarty et al. The ghc commentary. Checked in to the GHC CVS, available at http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/.
7. U. Klusik. *An Efficient Implementation of the Parallel Functional Language Eden on Distributed-Memory System.* PhD thesis, University of Marburg, 2002. In prep.
8. U. Klusik, Y. Ortega-Mallén, and R. Peña Marí. Implementing Eden – or: Dreams Become Reality. In *IFL'98 – Intl. Workshop on the Impl. of Funct. Lang.*, LNCS 1595, pages 103–119. Springer, 1999.
9. H.-W. Loidl, U. Klusik, K. Hammond, R. Loogen, and P. Trinder. GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster. In *SFP'00 – Scottish Funct. Progr. Workshop*, Trends in Functional Programming, Vol. 2, pages 39–52. Intellect, 2000.
10. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing.* Springer, 2002, to appear.
11. S. Marlow, S. Peyton Jones, and A. Reid. The stg runtime system (revised). Part of the GHC distribution.
12. S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *JFP*, 2(2):127–202, 1992.
13. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, U.K., March 1993. http://www.dcs.gla.ac.uk/fp/papers/grasp-jfit.ps.Z.
14. S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at `http://www.haskell.org/`.
15. S. Peyton-Jones and S. Marlow. The new ghc/hugs runtime system. Unpublished, available at www.research.microsoft.com/~simonpj/Papers/new-rts.ps.gz.
16. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, 1998.
17. P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Language Design and Implementation*, pages 78–88. ACM Press, May 1996.