Testing Speculative Work in a Lazy/Eager Parallel Functional Language*

Alberto de la Encina, Ismael Rodríguez, and Fernando Rubio

Facultad Informática. Universidad Complutense de Madrid
C/. Prof. José García Santesmases, E-28040 Madrid. Spain.
{albertoe,isrodrig,fernando}@sip.ucm.es

Abstract Eden is a parallel extension of the functional language Haskell. Eden inherits from Haskell its *laziness*, which allows it to avoid unnecessary computations. However, in order to enable the parallel execution of processes in Eden, this feature must be disabled when new processes are instantiated. Hence, any newly created process can be *speculative*, as it is not known whether the computations it performs will actually be required for the overall computation. Therefore, the performance of a program may be affected by the unneeded speculation. In this paper we present a framework to compare the speculated computations of an Eden program with the computations it actually requires. Thus, the programmer is provided with a profiling tool allowing him to produce better programs where speculative work fits better the actual necessities.

1 Introduction

Parallel programming faces several specific challenges that are not met in sequential programming. The programmer of a sequential program defines a computation in terms of some subcomputations, and the coordination of them is trivially achieved because the order of subcomputations is implicitly given. However, the coordination of subcomputations in parallel programs increases their complexity. In this sense, the *functional* paradigm provides some advantages for the programmer. In particular, parallel functional languages are endowed with useful abstraction mechanisms like function composition and higher-order functions. The higher-order programming level provided by them allows to define the coordination of subcomputations in terms of the same constructions used in the rest of the program, which enables the definition and use of skeletons [2,3,7] to develop simpler parallel programs. Besides, since functional programs do not have *state*, side-effects are eliminated. So, the dependencies between processes are limited to obtaining the arguments needed to execute each function. These features ease the coordination issues and allow to define them in a natural way.

Several parallel functional languages have been proposed (see e.g. [20,7,17,19]). Among them, Eden has the interesting characteristic of requiring relatively low

* Work partially supported by the MCYT project TIC2003-07848-C02-01, the JCCLM project PAC-03-001, and the Marie Curie project MRTN-CT-2003-505121/TAROT.

programming effort to create programs with acceptable speedups (see [12]). Its main advantage is that it combines high-level constructions to simplify the development of parallel programs, and some controlled low-level constructions to allow increasing the efficiency. Eden extends the (lazy evaluation) functional language Haskell [15] by adding syntactic constructions for defining and instantiating processes. As a Haskell extension, Eden applies the *laziness* for deciding the computations to be executed in each moment. That is, a computation is performed only after it is detected that the result of that computation is required for continuing another computation that is already initiated. Let us note that pure laziness implies *sequential computation*. So, in order to allow parallel computations, Eden creates new processes *eagerly*. Moreover, any newly created process is able to perform computations in parallel before its creator actually demands the result for continuing its execution. This feature, which is necessary for enabling parallelism, may cause that a program performs some computations that turn out to be unneeded. In fact, Eden processes are *speculative*: They perform computations under the assumption that they will actually be needed.

The uncontrolled speculation may be a source of inefficiency in parallel programs. In order to achieve a better use of resources and a higher performance, the programmer should be provided with a measure of the *unnecessary speculation* of a program. In this paper we present a method for comparing the speculative computations and the computations actually needed in an Eden program. Basically, the method consists in comparing the data actually needed by a process and the speculative data evaluated by processes launched by this process.

Unfortunately, making a functional program to show the results of partial computations in some points is not easy. Let us remind that, contrarily to an imperative program, a functional program does not have *state*. Thus, the observation of partial computations cannot be based on observing how some variables change, because variables do not exist in functional environments. Besides, due to the laziness of Eden, the execution of a computation may turn out to be *unnecessary*, but a simple *observation* (e.g., writing a result in the screen or in a file) could create a false demand on such unneeded computation. Hence, observations must be defined in such a way that they produce a (neutral) result that is actually required only in the same situations as if the observations were not introduced. We will address this issue by using and extending *Hood* (*Haskell Object Observation Debugger* [5]). This tool allows a programmer to observe the behavior of a Haskell program by inserting some calls to an *observation function* in the program. The observation function records the value returned by a function in some point of the program, but without creating extra demand. These functions will be the basis of our method to compare the useful speculation and the actual speculation in an Eden program.

The rest of the paper is structured as follows. In the next section we sketch the Eden language. Then, in Section 3 we present the observation constructions of Hood. Next, in Section 4 we present our method to assess the unnecessary speculation in Eden programs. A case study is shown in Section 5, and related work in Section 6. Section 7 contains our conclusions and lines of future work.

2 The Eden Language

Eden [7,13] extends the lazy functional language Haskell [15] by adding syntactic constructs to explicitly define and instantiate processes. It is possible to define a new *process abstraction* `p` by using the following notation that relates the inputs and the outputs of the process: `p = process x -> e` , where variable `x` will be the input of the process, while the behavior of the process will be given by expression `e`. Process abstractions can be compared to functions, the main difference being that the former, when instantiated, are executed in parallel.

Process abstractions are not actual processes. To really create a process, a *process instantiation* is required. This is achieved by using the predefined infix operator `#`. Given a process abstraction and an input parameter, it creates a new process and returns the output of the process. Each time an expression `e1 # e2` is evaluated, the instantiating process will be responsible for evaluating and sending `e2`, while a new process is created to evaluate the application `(e1 e2)`.

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 not yet available inputs are temporarily suspended. This is the only way in which Eden processes synchronize. Notice that process creation is explicit, but process communication (and synchronization) is completely implicit.

In contrast to most parallel functional languages, Eden also includes high-level constructions (not shown in the paper) both for developing *reactive* applications and for dynamically establishing direct connections between any pair of processes. This allows handling *low-level* parallel features that cannot be used in conventional functional languages. Thus, Eden provides an intermediate point between very high-level parallel functional languages (whose performance use to be poor), and classical parallel languages (which do not allow using high-level constructions). We do not claim that Eden can obtain optimal speedups, but it can obtain quite *acceptable* speedups with small programming effort (see e.g. [13,12]).

Eden's compiler (see `http://www.mathematik.uni-marburg.de/inf/eden`) has been developed by extending the most efficient Haskell compiler (GHC [14]). Hence, Eden's compiler reuses GHC's capabilities to interact with other programming languages. Thus, Eden can be used as a coordination language, while the sequential computation language can be, for instance, C.

To easily port the compiler to different architectures, the runtime system works on top of a message passing library (the user can choose PVM or MPI).

Eden Skeletons Process abstractions in Eden are not just annotations, but first class values which can be manipulated by the programmer (passed as parameters, stored in data structures, and so on). This facilitates the definition of skeletons as higher order functions. Next, we illustrate, by using simple examples, how skeletons can be written in Eden. More complex skeletons can be found in [13,18].

The most simple skeleton is `map`. Given a list of inputs `xs` and a function `f` to be applied to each of them, the sequential specification in Haskell is as follows:

```
map f xs  =  [f x | x <- xs]
```

that can be read as *for each element* `x` *belonging to the list* `xs`*, apply function* `f` *to that element*. This can be trivially parallelized in Eden. In order to use a different process for each task, we will use the following approach:

```
map_par f xs =  [pf # x | x <- xs] `using` spine
  where pf = process x -> f x
```

The process abstraction `pf` wraps the function application `(f x)`. It determines that the input parameter `x` as well as the result value will be transmitted through channels. The `spine` strategy (see [21] for details) is used to eagerly evaluate the spine of the process instantiation list. In this way, all processes are immediately created. Otherwise, they would only be created on demand.

Let us remark that it is not necessary to explicitly use constructions for synchronizing the processes. The main process initially sends a task to each of the *worker* processes of the `map_par`. Afterwards, as soon as any of the workers finishes its assignment, it automatically sends the result to the main process (by using PVM or MPI messages). When the main process has received all the results that it needs, it finishes the computation.

`map_par` is an essential primitive skeleton used to eagerly create a set of independent processes, but it can be easily improved by reducing the number of processes to be created. In a `map_farm` the number of processes to be created is fixed (for instance, it can be the number of processors), and tasks are evenly distributed into processes. The implementation firstly distributes the tasks among the processes, producing a list of lists where each inner list is to be executed by an independent process. Then, it applies `map_par`, and finally it collects the results joining the list of lists of results into a single list of results. Notice that, due to the laziness, these three tasks are not done sequentially, but in interleaving. As soon as any worker computes one of the outputs it is computing, it sends this subresult to the main process, and it goes on computing the next element of the output list. Notice that the communications are asynchronous, so that it is not necessary to wait for acknowledgments from the main process. When the main process has received all the needed results, it finishes the computation. The Eden source code of this skeleton is shown below, where not only the number `np` of processors but also the distribution and collection functions (`unshuffle` and `shuffle` respectively) are also parameters of the skeleton:

```
map_farm np unshuffle shuffle f xs
  = shuffle (map_par (map f) (unshuffle np xs))
```

Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle (unshuffle np xs)) == xs`.

Let us remark that developing skeletons in Eden is relatively easy. Due to the lack of space, we have only shown the simplest examples, but many others have already been implemented. Details about them can be found in [13].

3 Basic Hood

In this section we show the basic ideas behind Hood. The interested reader is referred to [5,4] for more details about it.

When debugging programs written in an imperative language, the programmer can explore not only the final result of the computation, but also the intermediate values stored in the variables being used by the program. Moreover, it is simple to trace how the value of each variable changes along time.

Let us remark that debugging lazy functional code has two main difficulties. First, lazy functional languages do not contain variables whose value change along time and that can be traced as in imperative languages. Second, introducing observations can modify the order of evaluation, affecting to the overall computation.

Fortunately, Hood allows the programmer to observe something similar to what can be observed in imperative environments. In fact, Hood allows the programmer to observe any intermediate structure appearing in a program. Moreover, we can also observe the evolution in time of the evaluation of the structures under observation.

In order to illustrate what kind of observations can be obtained by using Hood, let us consider an example. It will be complex enough to highlight important aspects of Hood, but also relatively simple to be easily understandable without requiring knowledge about Haskell. Given a natural number, the following Haskell function returns the list of digits of that number:[1]

```
natural :: Int -> [Int]
natural = reverse
          . map (‘mod‘ 10)
          . takeWhile (/= 0)
          . iterate (‘div‘ 10)
```

That is, `natural 3408` returns the list `3:4:0:8:[]`, where `[]` denotes the empty list and `:` denotes the list constructor. Note that, in order to compute the final result, three intermediate lists where produced in the following order:

```
-- after iterate
3408:340:34:3:0:_
-- after takeWhile
3408:340:34:3:[]
-- after map
8:0:4:3:[]
```

[1] The first line of the definition only provides the type declaration of the function: given an integer it returns a list of integers. The other four lines define the sequence of functions to be applied to obtain the overall effect, being `reverse` the last one to be applied. The higher-order function `iterate` applies infinite times the first function it receives. For instance, applying `iterate (+3) 1` returns the infinite list `1:4:7:10:13:...`

Notice that the first intermediate list is infinite, although only the first five elements are computed. As the rest of the list does not need to be evaluated, it is represented as _ (the underscore char).

By using Hood we can annotate the program in order to obtain the output shown before. In order to do that, we have to use the `observe` combinator that is the core of Hood. The type declaration of this combinator is: `observe :: String -> a -> a` . From the evaluation point of view, `observe` only returns its second value. That is, `observe s a = a`. However, as a side effect, the value associated to `a` will be squirrelled away, using the label `s`, in a file that will be analyzed after the evaluation finishes. It is important to remark that `observe` returns its second parameter in a completely lazy, demand driven manner. That is, the evaluation degree of `a` is not modified by introducing the observation, in the same way that it is not modified when applying the identity function `id`. Thus, as the evaluation degree is not modified, Hood can deal with infinite lists like the one appearing after applying `iterate (‘div‘ 10)`.

If we consider again our previous example, we can observe all of the intermediate structures by introducing three observations as follows:

```
natural :: Int -> [Int]
natural = reverse
          . observe "after map"       . map (‘mod‘ 10)
          . observe "after takeWhile" . takeWhile (/= 0)
          . observe "after iterate"   . iterate (‘div‘ 10)
```

After executing `natural 3408`, we will obtain the desired result. Hood does not only observe simple structures like those shown before. In fact, it can observe anything appearing in a Haskell program, including functions. For instance,

```
observe "length" length (4:2:5:[])
```

will generate the following observation:

```
-- length
   { \ (_:_:_:[]) -> 3 }
```

That is, we are observing a function that returns the number 3 (without evaluating the concrete elements appearing in the list) when it receives a list with three elements. Let us remark that it is only relevant the number of elements, but not the *concrete* elements. That is, the observation mechanism detects that the laziness of the language will not demand the concrete elements to compute the overall output. As it can be expected, higher-order functions can also be observed, but we do not show it due to lack of space.

4 Testing Speculation by Parallelizing Hood

In this section we present our method to assess the speculation of a parallel Eden program. The method is based on using the observation functionalities provided

by Hood in specific points of the program under assessment. Let us recall that the application of `observe` to a term returns only the (partial) evaluation of the term that is actually *required* by other subcomputations in the context where `observe` is invoked. Hence, it gives us the (partial) term that is demanded in the context of the observation. When a process instantiates another process, the former process demands the computation of a term from the latter (from now on, *invoker* and *instantiated* processes, respectively). In order to perform our analysis, we need to consider the evaluation of this term at two different points. On the one hand, the observation of the term required by the invoker (in the context of the invoker) gives us the true necessities of the program at this point. On the other hand, the observation of the term constructed by the instantiated process (in the context of instantiated process) gives us the result of the speculated work performed by the instantiated process. By comparing both values, we can assess the amount of unnecessary speculation performed by the program at this point. In fact, if we obtain not only the final values of the term at both sides but also the order in which each part of the term is calculated, then we can infer not only the amount of unnecessary work but also the relative speeds of both parts. Hence, we can enrich the profiling capabilities of Eden.

4.1 A Simple Example

Let us consider a naïve but illustrative example. The following process generates the (infinite) list of primes greater than or equal to a given input number `n`. The process receives a natural number `n` as input, and produces a (potentially infinite) list of primes `outputs`:

```
pprimes = process n -> outputs
  where outputs = generatePrimes n
generatePrimes x = if (isPrime x) then x : restOfPrimes
                                  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

The list of primes `outputs` is obtained by calling function `generatePrimes` with parameter `n`. Given any parameter `x`, this function firstly computes the (infinite) list of all the rest of primes, that is, the list of all primes greater than or equal to `x+1` (as we will see below, depending on the necessities of other external computations, only a finite part of the list might be computed). Then, `x` is added at the head of the list if `x` is prime or it is ignored otherwise.

Let us suppose that we are interested in using this process to obtain the shortest list of consecutive primes (greater than or equal to `initialNumber`) such that the multiplication of its elements is greater than or equal to a given minimal threshold `threshold`.[2] For example, given `initialNumber=2` and `threshold=26`, we wish to obtain `[2,3,5]` since $2 * 3 * 5 = 30$. The function `myComputation` performs this task:

[2] A similar functionality will be required in Section 5 to implement the `LinSolv` algorithm. In particular, it will be required to apply the Chinese Remainder Theorem.

```
myComputation initialNumber threshold = take neededNumber primes
  where primes   = pprimes # initialNumber
        products = scanl (*) 1 primes
        neededNumber = length (takeWhile (< threshold) products)
```

Let us explain how function `myComputation` works. The term `primes` represents the list of *all* primes from `initialNumber` on, and it is computed by instanting a new process that executes function `pprimes` with parameter `initialNumber`. Fortunately, the rest of expressions used in the context of function `myComputation` will only demand a finite amount of elements of `primes`. Thus, the new process will not compute new primes forever. This is so because when the invoker process finishes the computation of function `myComputation`, the runtime system automatically terminates the instantiated process. The term `products` performs the multiplication of all elements in list `primes` by using function `scanl`. Function `scanl` applies a given binary operator to all the elements in a list. This is done by applying the function to each element in order and cumulating the partial result from each element to the next. An initial cumulated value is given as parameter. Function `scanl` returns a new list where the elements are the cumulated values after each element in the input list is applied. For instance, `scanl (*) 1 [2,3,5]` returns the list `[1,2,6,30]`. The term `neededNumber` computes the number of elements in `products` that are below `threshold`. Since the ith element in `products` provides the multiplication of the first i primes of `primes`, `neededNumber` gives us the number of elements needed in list `primes`. The number `neededNumber` is calculated by taking the list of the elements of `products` below `threshold` and computing its length. Finally, the output of `myComputation` is given by taking `neededNumber` elements from the list `primes`.

Let us remark that the aim of the previous program is not to provide an efficient parallel solution, but to serve as a simple but illustrative basis to develop a first approach to our method. In particular, we will use the previous example to study the speculative work performed by the instantiated process to compute the function `pprimes`. Although this process is devoted to produce an infinite list of numbers, not all of them will be actually required. However, we do not know in advance how many elements of the list will be actually required. We are interested in comparing the amount of primes used by the process that executes the function `myComputation` with the amount of primes computed by the process that runs the function `pprimes`. In fact, all primes calculated by `pprimes` that are higher than the last prime used by `myComputation` are the result of the *unnecessary* speculated work, because these primes are useless for the program necessities. Let us note that a programmer is not likely to properly assess the amount of unnecessary speculative work of this program in advance, because the time required by each process to perform each operation depends on several uncontrollable factors. Actually, both involved processes will race each other to perform their computations. Hence, if their relative speed is unbalanced (in any of both senses) then the overall performance of the program will fall.

Next we show how the previous program can be modified to provide the required information. The modification will be based on introducing observations

that will report the computation/use of the list of primes in both processes. First, let us consider the instantiated process. In order to observe the list of primes that is produced by this process, we just need to observe the list that it returns to its invoker. The term to be returned is now replaced by a new expression. It returns the same value to the invoker, but only after any change on the evaluation of `outputs` is properly reported to a log file with a suitable tag (`outsFromProcess`). So, though the introduction of the observation is innocuous for the overall computation, the required profiling information will be obtained:

```
pprimes = process n -> (observe "outsFromProcess" outputs)
  where outputs = generatePrimes n
```

Let us consider the instantiated process. In order to minimize the number of requests between invoker processes and instantiated processes, any value computed by an instantiated process is immediately sent to the invoker in Eden. This means that all primes computed by the instantiated process are locally available for being used by the invoker. However, let us note that the laziness is locally applied in each process in Eden. Hence, the number of primes *obtained* by the invoker (i.e., actually *taken* from those received by the communication channel) perfectly matches its necessities, and no unneeded prime is used. So, by observing the list of primes obtained by the invoker from the instantiated process, we can calculate the true necessities of this process. Any change in this list will be reported in a file with the tag `insFromProcess`:

```
myComputation initialNumber threshold = take neededNumber primes
 where primes = observe "insFromProcess" (pprimes # initialNumber)
       products = scanl (*) 1 primes
       neededNumber = length (takeWhile (< threshold) products)
```

After running the program using `327` and `49472453` as inputs in a two-processors environment, our observations inform us that `outsFromProcess` has 37 entries, while `insFromProcess` has only 4. That is, 33 unnecessary primes where computed by the auxiliary process in our naïve example.

4.2 General Scheme

In the previous example, the speculative work was performed by the instantiated process. However, this could be the other way around: The invoker process instantiates a new process and afterwards produces some values that this new process may or may not need. The values demanded by the instantiated processes are its *parameters*. In this case, the instantiated process takes the values computed by the invoker as long as it needs them, and the speculative work is performed by the invoker process. Let us note that the application of our method to this case is similar to that shown in the previous example. Actually, the method developed in the previous example can be generalized to deal with any scenario where the speculative work of some processes has to be assessed. Next we present a redefinition of the basic Eden constructors. This redefinition

performs all the required tracing issues in such a way that the programmer can forget any details concerning observations: He must just instantiate the processes he wants to analyze by calling the functions provided by the new constructors and introducing his function as parameter. Then, the system automatically reports any change on both its input and its output. By applying the new observation constructors to both an invoker and an instantiated process, all the needed information will be properly reported. Thus, if we want to observe the inputs and outputs of a process that computes a given function `f` then, instead of directly using `f`, we will call the following function `processObs` using `f` as parameter:

```
processObs f = process ins -> (observe "outsFromProcess" outs)
          where outs = f ins'
                ins' = observe "insToProcess" ins
```

The previous function defines a process with input `ins`. In order to observe the data that this new process receives from its creator and it actually *requires*, this parameter is observed by the second observation in the previous definition, labelled by `insToProcess`. After function `f` is normally applied to the input, the output `outs` is obtained. The observation of this term (first observation, labelled by `outsFromProcess`) reports the data this process transmits to its creator.

The previous function allows us to observe the treatment of inputs and outputs of the instantiated process. Similarly, we need a new functionality to observe the behavior of the invoker process. Next we redefine the process instantiation operator to include the observation capabilities. The new operator, based on the standard operator `#`, is `##`:

```
p ## actualParameters =
  observe "insFromProcess"
          (p # (observe "outsToProcess" actualParameters))
```

The new operator allows any process to instantiate a new process by using the standard one. Besides, two observations are introduced to report the inputs and outputs that the invoker process exchanges with the new process. Observations labelled by `insFromProcess` report the data that the invoker receives (and actually requires) from the newly instantiated process. Observations labelled by `outsToProcess` report the data that is sent from the invoker to the instantiated process (regardless of whether the instantiated process requires them).

The use of both new constructors leads to the general scheme depicted in Figure 1. By combining the new process abstractions `processObs` and `##` we obtain four relevant data. These data provide us with two critical measures concerning the usefulness of the speculation at this point of the program. On the one hand, the difference between `outsFromProcess` and `insFromProcess` gives us how much unnecessary speculative work was done by the new process. On the other hand, the difference between `outsToProcess` and `insToProcess` provides us a measure to know how much unnecessary speculative work was performed by the process creating the new instantiation.

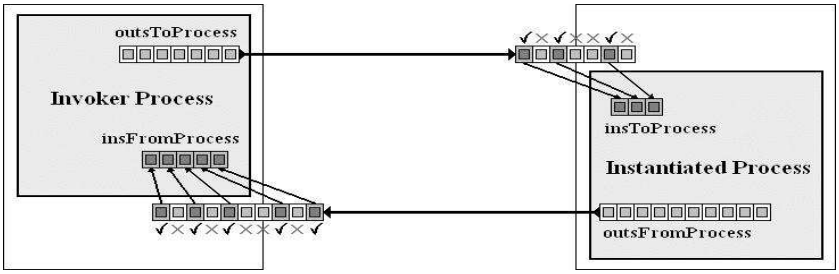

Figure 1. Invoker and Instantiated Processes

Let us note that the definitions of `processObs` and `##` could be trivially extended to include extra parameters representing the strings that want to be used for marking the inputs and outputs of the processes. Moreover, the framework can be easily applied to other general schemes and programming structures in Eden. In particular, all the skeletons defined in the Eden library can be trivially rewritten in terms of the new process abstraction and process instantiation operators. Hence, they inherit the capability to test the amount of speculative work.

5 Case Study: LinSolv

The `linSolv` algorithm finds an exact solution of a linear system of equations of the form $Ax = b$ where $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n, n \in \mathbb{N}$. In contrast to more common numerical algorithms, which usually produce an approximate solution over floating point numbers for a given accuracy, the algorithm presented here finds an exact solution and works over arbitrary precision integers.

To find an exact solution for a given system of equations, `linSolv` uses a *multiple homomorphic images* approach [9]. This is a common computer algebra approach and consists of the following three stages: (1) map the input data into several homomorphic images; (2) compute the solution in each of these images; and (3) combine the results of all images to a result in the original domain.

This structure is particularly useful for operations on arbitrary precision integers. In this case the original domain is $\mathbb{Z}$, the set of all integer values, and the homomorphic images are $\mathbb{Z}$ modulo p, written $\mathbb{Z}_p$, with p being a prime number. If the input numbers are very big and each prime number fits into one machine word the basic arithmetic in the homomorphic images is cheap because fixed precision arithmetic can be used. Only in the combination phase, when applying a fold-based Chinese Remainder Algorithm (CRA) (see [10]), expensive arbitrary precision arithmetic has to be used to construct the result values.

Details about the implementation of `linSolv` in Haskell can be found in [12]. In brief, the main part to be parallelized consists in solving each of the homomorphic images, whose basic definition is: `xList = map get_homSol primes`

```
xList_all = map_farm get_homSol primes

xList = filter lucky xList_all
```

Figure 2. Parallel `linSolv` (Eden speculative version)

```
xList_all = map_rw get_homSol primes

xList = filter lucky xList_all
xList_unlucky = filter (not.lucky) xList_all

(p_needed, p_spec) = splitAt ( 1 + toInt noOfPrimes) primes
primes' = p_needed ++ (additional xList_unlucky p_spec)

additional :: [Integer] -> [Integer] -> [Integer]
additional xs ys = zipWith (\ x y -> y) xs ys
```

Figure 3. Parallel `linSolv` (Eden conservative version)

where `primes` is an infinite list of primes, and `get_homSol` solves the system modulo a given prime. Thus, the basic parallel structure of the algorithm consists in performing all computations in the homomorphic images in parallel. It uses LU-decomposition followed by forward and backsubstitution to compute the solution `pmx` in the homomorphic image [16]. From a speculation point of view, the main difficulty in the parallelization is that we have to make sure that new results are computed if primes turn out to be "unlucky", i.e. if the determinant of the input matrix A in the homomorphic image generated by this prime number is zero. Let us remark that this is similar to the naïve example we showed in the previous section. However, in this case the task of the processes is not only to create primes, but also to solve a linear system modulo that prime.

Even though the situation now is more complex than in the example of the previous section, our strategy to check how much useless work is done is the same. As we have to solve the linear system modulo several prime numbers, the most obvious parallel scheme is to use a `map_par` scheme as shown in Section 2, so that an independent process is created for each prime. However, as we commented in Section 2, it is better to use a `map_farm` to avoid creating too many processes. So, in our first approach (shown in Figure 2) we just replaced the top level `map` by its parallel counterpart `map_farm`. Unfortunately, when using the `map_farm` version that includes observations, our tools detected a quite big amount of useless work, that reduced considerably the overall speedup.

As a second approach, to avoid the potential waste of resources due to speculation we used a *conservative version* as shown in Figure 3. In this version the prime numbers are divided into those known to be needed (`p_needed`) and those which are only needed if some of the earlier primes are unlucky (`p_spec`). The function `additional` adds for each unlucky prime a new prime number to the task list `primes'`. Note in the definition of `additional` that, due to the demand-driven evaluation, the availability of unlucky primes in `xs` triggers the generation of one result element in `ys`. With this conservative version, the amount of useless

```
xList_all = map_rw get_homSol primes
xList = filter lucky xList_all
```

Figure 4. Parallel `linSolv` (Eden semi-speculative version)

work was zero. Unfortunately, this does not necessarily implies optimal speedups. The problem is that we had avoided useless work, but at the cost of forcing processes to stop for a while each time they finish a task. So, tough the speedups were better than before, there was still free room for improvement.

Finally, we used a third solution where speculation was restricted but not completely avoided. In this sense, we used a variation of the task farm skeleton as outlined in Section 2. More specifically, we used the replicated workers paradigm. A manager and a set of worker processes are created, and two tasks are initially released to each of the workers. As soon as any worker finishes a task, it sends the result to the manager, and a new task is delivered to the worker. The computation in the manager is demand-driven and triggered by the availability of result values. As soon as the manager has all the needed results it terminates all the worker processes. Notice that in this *semi-speculative version* the workers may be working speculatively on useless tasks, but only when the useful tasks have already been consumed and hence the degree of speculation is tightly limited. More details about the replicated workers skeleton can be found in [8]. Figure 4 shows the Eden code for the semi-speculative version of `linSolv`. The only modification to the sequential code is the use of a parallel replicated workers map (`map_rw`) instead of a sequential map over the infinite list of primes. By using this new version, only a few useless messages where sent. That is, the speculation was actually controlled.

Finally, we illustrate the speedups that can be obtained with Eden, by running the semi-speculative version on a concrete parallel machine. We run the experiments on a 32-node Beowulf cluster consisting of workstations with a 533 MHz Celeron processor, 128 Kb cache and 128 MB of DRAM. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142 μs, measured under PVM 3.4.2. The sequential runtime in this environment was 491.7s. In this environment, an acceptable speedup of 14 is achieved with 16 nodes. For the input data used in these measurements 45 useful and 39 unlucky primes are generated. This leads to a total of 45 top level threads, one for each homomorphic image.

6 Related Work

In addition to Hood, during the last years there have been several proposals for incorporating execution traces to sequential lazy functional languages. In particular, we can highlight the work done with Hat, HsDebug, and the declarative debuggers Freja and Buddha. The approaches followed in each of them are quite different, both from the point of view of the user of the system and

from the implementation point of view. From the user point of view, Freja and Buddha are question-answer systems that directs the programmer to the cause of an incorrect value, while Hat allows the user to travel backwards from a value along the redex history leading to it. The interested reader can found a detailed comparison between Freja, Hat and Hood in [1].

Regarding parallel functional profilers, we can highlight GranSim [11] and its derivatives (GranSP and GranCC). GranSim is a GpH [20] simulator. This system provides an accurate and flexible way of studying the dynamic behaviour of GpH programs. It supports extensive tuning of the simulated architecture, having parameters such as number of processors, communication latencies, and others. Additionally, an ideal simulation mode allowing an unlimited number of processors is provided.

Paradise [6] is an adaptation of GranSim to deal with Eden programs. Currently, Eden programers can obtain feedback from Paradise in order to improve the performance of their programs. In this sense, they can detect bottlenecks in the distribution of work. However, it lacks the possibility to check the amount of speculation done by the programs. Thus, the approach presented in this paper constitutes a complement to the profiling capabilities presented in Paradise.

7 Conclusions and Future Work

In this paper we have extended the parallel functional language Eden with capabilities to test how much useless work has been performed in a given execution. Since the core sequential part of Eden uses lazy evaluation while some parallel constructions of it require the use of strict evaluation, we have designed a tool to compare how much data was actually needed (known because of the lazy part of the language) and how much data was actually transmitted (known because of the eager parallel part of the language).

We have rewritten the basic constructions of the language to include facilities for testing the amount of speculative work, and we have also rewritten skeleton libraries so that we can test the amount of speculation of any program written by using skeletons. In fact, we have tested the tool with a concrete case study that consists in solving a linear system of equations.

Currently, our tools only provide textual information about the amount of speculation, and this is the reason why we do not show examples of outputs of the tool in this paper. However, we are working on the implementation of a graphical interface to show not only the amount of speculation, but also the evolution in time of the speculation of the program. This can be done by recording also time-information about the observations. Thus, the implementation of this graphical tool and the application of the framework to a wider set of examples constitute our current lines of work.

References

1. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs.

In *IFL'00*, LNCS 2011, pages 176–193. Springer-Verlag, 2001.
2. M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations*. MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.
3. M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30:389–406, 2004.
4. A. Encina, L. Llana, and F. Rubio. Formalizing the debugging process in Haskell. In *International Conference on Theoretical Aspects of Computing, ICTAC'05*, LNCS 3722, pages 211–226. Springer-Verlag, 2005.
5. A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*. Tech. Rep. University of Nottingham, 2000.
6. F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *Scottish Functional Programming Workshop, SFP'99*, pages 11–19. Intellect, 2000.
7. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *Implementation of Functional Languages, IFL'00*, LNCS 2011, pages 71–88. Springer-Verlag, 2001.
8. U. Klusik, R. Peña, and F. Rubio. Replicated workers in Eden. In *Constructive Methods for Parallel Programming, CMPP'00*, pages 143–164. Nova Science, 2000.
9. M. Lauer. Computing by homomorphic images. In *Computer Algebra — Symbolic and Algebraic Computation*, pages 139–168. Springer-Verlag, 1982.
10. J. D. Lipson. Chinese remainder and interpolation algorithms. In *Symposium on Symbolic and Algebraic Manipulation, SYMSAM'71*, pages 372–391. Academic Press, 1971.
11. H. W. Loidl. Gransim user's guide. Department of Computing Science. University of Glasgow, 1996.
12. H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.
13. 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*, pages 95–128. Springer-Verlag, 2002.
14. S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming, ESOP'96*, LNCS 1058, pages 18–44. Springer-Verlag, 1996.
15. S. L. Peyton Jones and J. Hughes. Report on the programming language Haskell 98. Technical report, February 1999. `http://www.haskell.org`.
16. W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, chapter LU Decomposition and Its Applications. Cambridge University Press, 2nd Edition, 1992.
17. R.F. Pointon P.W. Trinder, H.W. Loidl. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.
18. F. Rubio and I. Rodríguez. A parallel framework for computational science. In *International Conference on Computational Science, ICCS'03*, LNCS 2658, pages 1002–1011. Springer-Verlag, 1998.
19. N. Scaife, Horiguchi S., G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *J. Functional Programming*, 15(4):615–650, 2005.
20. P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Programming Language Design and Implementation, PLDI'96*, pages 79–88. ACM Press, 1996.
21. P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.