Repeating History: Execution Replay for Parallel Haskell Programs

Henrique Ferreiro[1], Vladimir Janjic[2], Laura Castro[1], and Kevin Hammond[2]

[1] Department of Computer Science, University of A Coruña, Spain
{hferreiro,lcastro}@udc.es
[2] School of Computer Science, University of St Andrews, United Kingdom
{vj32,kh}@cs.st-andrews.ac.uk

Abstract. Parallel profiling tools, such as ThreadScope for Parallel Haskell, allow programmers to obtain information about the performance of their parallel programs. However, the information they provide is not always sufficiently detailed to precisely pinpoint the cause of some performance problems. Often, this is because the cost of obtaining that information would be prohibitive for a complete program execution. In this paper, we adapt the well-known technique of *execution replay* to make it possible to simulate a previous run of a program. We ensure that the non-deterministic parallel *behaviour* of the application is properly emulated while the deterministic *functional* code is run unmodified. In this way, we can gather additional data about the behaviour of a parallel program by replaying some parts of it with more detailed profiling information. We exploit this ability to identify performance bottlenecks in a quicksort implementation, and to derive a version that gives better speedups on multicore machines.

1 Introduction

Writing *correct* parallel programs in pure functional languages, such as Glasgow Parallel Haskell (GpH [10, 15]), is relatively simple, since the absence of side-effects means that it is not necessary to worry about some situations such as race conditions or deadlocks that can seriously complicate parallel programs written using more traditional techniques. However, writing *good* parallel programs, which will give good speedups on a wide variety of parallel architectures, is much harder. Understanding why a seemingly "perfect" parallel program does not perform the way the programmer expects can be difficult, especially in a lazy language like Haskell. Profiling can greatly help in understanding the performance of parallel programs. Current tools for profiling parallel functional programs, such as ThreadScope [6], allow the programmer to obtain some information about the behaviour of the parallel program. However, the information they give is often too low-level to pinpoint performance problems (in the case of ThreadScope), or their use can change the runtime behaviour of the original program (in the case of cost centre profiling).

In this paper, we describe how to adapt the well-known technique of *execution replay* [14] to allow us to do performance debugging of parallel functional

programs. Traditionally, execution replay has been used to debug imperative programs, and its essence is in replaying the execution of a program in order to reproduce the same state of the memory and registers as in the original execution. To the best of our knowledge, this paper represents both the first attempt to use this technique in the context of a lazy functional language, and its first adaptation for parallel performance debugging. With our implementation, the repeated execution of a program is *simulated* in a way that allows us to *i*) reproduce the conditions that led to the poor parallel performance and *ii*) make changes to the program execution in order to collect additional information about its runtime behaviour. In this way, we can dynamically tune the amount and type of profiling we do during the replay to get high-level profiling information without changing the runtime behaviour of the original program.

In particular, the paper presents the following novel research contributions:

- We describe the implementation of execution replay for the parallel programs written in the pure lazy functional language Haskell. In particular, we describe the smallest set of events from the program execution that needs to be recorded in order to reproduce the parallel behaviour of these programs. We subsequently present a simulator we built to replay the program execution using these events.
- We discuss how this technique can be used for performance debugging of Parallel Haskell programs.
- We present a use case, where execution replay is used to discover the performance bottleneck of a simple program (quicksort) which appears easy to parallelise, yet it is quite subtle to obtain good speedups.

2 Why is Parallel Functional Programming Hard?

The lack of explicit program flow, and the fact that a lot of things happen implicitly during the program execution, is both a blessing and a curse for parallel functional programmers, especially in a lazy language like Haskell. While it is arguably easier to write parallel programs in Haskell than in imperative languages (the programmer "just" needs to insert simple parallel annotations in the appropriate places in his code), discovering performance bottlenecks of such programs can be daunting. There are just so many things that can go wrong, and of which the programmer does not have explicit control. Consider, for example, a simple parallel implementation of the quicksort list-sorting algorithm:

```
psort :: Int → [Int] → [Int]
psort _          []  = []
psort parLevel l@(x : xs)
  | parLevel > 0 = hiSorted `par` loSorted `pseq` (loSorted ++ x : hiSorted)
  | otherwise    = seqSort l
  where (lo, hi)   = partition (<x) xs
        loSorted   = psort (parLevel − 1) lo
        hiSorted   = force (psort (parLevel − 1) hi)
```

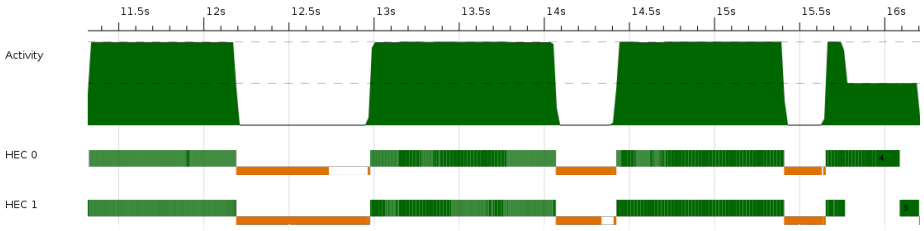

Fig. 1. ThreadScope profile of *psort*

The reasoning behind this attempt at parallelisation of quicksort is simple: after dividing the initial list l into its lower and higher parts (lo and hi) by using x as a pivot, we try to sort these two parts in parallel using the *par* combinator. Because of how laziness works, we use the function *force*[3] to make sure that each parallel thread completely evaluates its sublist. In addition, by using the *parLevel* parameter, we control the amount of parallelism generated, so that after a certain point in recursion is reached, the higher and lower parts of the list are sorted sequentially. In this way, we can tune the parallelism to get a small number of coarse-grained parallel threads. However, no matter what value for *parLevel* we chose, the speedups of *psort* are very poor, not even achieving a speedup of 2 in up to 8 cores.

In order to understand why this program gives a bad speedup, we can try to use ThreadScope to visualise what happens during its execution. Figure 1 shows the profile of the program. It shows a high-level overview of the threads activity on both cores. The solid rectangles indicate that a thread is running, the little marks in between and the rectangles not reflected in the *activity* area indicate garbage collection. Blank space indicates that the core is idle. We can zoom in on specific parts of the execution and obtain low-level information such as individual thread identifiers, some information about thread blockage, or garbage collection requests.

From the profile above, we can observe that one performance bottleneck lies in the three long garbage collection phases, where no useful work is performed. An additional problem seems to be the serialisation towards the end of the execution, where only one thread at a time is doing evaluation. However, ThreadScope does not provide us any hints about where do these problems come from, e.g. what data ends up being collected in these long garbage collection phases, or what part of the program is responsible for the final sequential phase. Based on the knowledge of the runtime behaviour of the language, we can speculate that the serialisation comes from the linear behaviour of the $+\!\!+$ operator, which traverses both lists sequentially. However, we cannot know for sure.

[3] $force :: NFData\ a \Rightarrow a \rightarrow a$ returns its argument after forcing its evaluation to normal form.

As we can see from the previous example, even though the information that we obtain using ThreadScope is valuable, it is too low-level to allow for proper understanding of the parallel performance of the program. What is really needed is much more detailed, high-level information about the runtime behaviour, ideally linking the parallel events from the ThreadScope profile to the source expresions they are related to. In the example above, knowing which expressions are being evaluated in the final sequential phase, and which ones are responsible for the great amount of wasted memory would be a first step into fixing the performance problems of this program. We come back to this problem in Section 5.

Obtaining the information required for performance debugging using existing infrastructure and tools would require either recording a huge amount of additional information, which could then be processed offline, or rerunning the program multiple times with different profiling options enabled. In both cases, the runtime behaviour (such as scheduling and communication between threads) of the original program might change, making the profiling data useless and making it very hard to reproduce the problem that is being debugged. Our solution to the problem is to reproduce the original execution of the program without changing its runtime behaviour while, at the same time, dynamically adjust the level and type of profiling information that is gathered during the execution. The way we did this was to adapt the technique of execution replay to parallel functional programs (and Parallel Haskell in particular).

3 Execution Replay for Parallel Haskell

Execution replay [14, 2] is a debugging technique in which a programmer records the execution trace of a program and then uses that trace to replay it step by step. The trace of the program encapsulates the whole state of the system as it changes throughout the execution. When replaying, the programmer is able to inspect the state of the program (e.g. variables, registers, stack) as it was at each step of the original execution.

Execution replay consists of two distinct phases (see Figure 2):

- a *logging phase* where, during the original program execution, enough information is logged so that the execution can be replayed and
- a *replaying phase* where the original program execution is replayed, using the logged information.

Our main goal is to use this technique to investigate performance bottlenecks of parallel programs, written in purely functional programming languages. This has an important consequence in that we do not have to be concerned with replaying *exactly the same* execution as the original one. It will suffice if the replayed execution is "similar enough" to the original one, such that both have the same parallel behaviour. We can, therefore, see a replay as a simulation of the execution where the threads created and the interactions between them are the same as in the original execution, and where other details of the execution may differ. This flexibility allows us to introduce changes in the program which

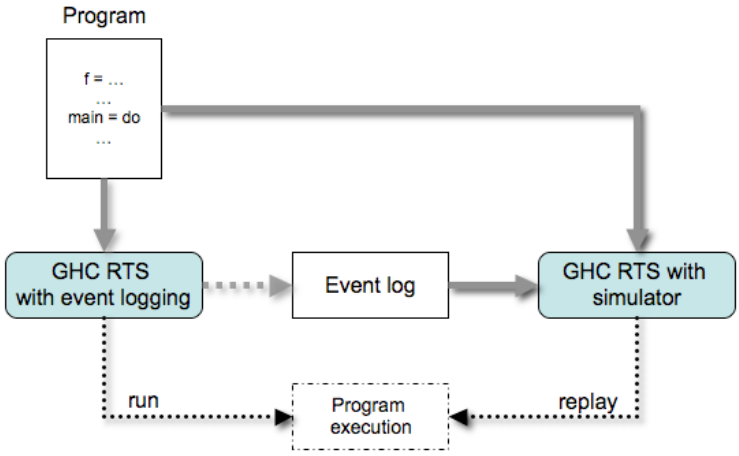

Fig. 2. Execution replay in Parallel Haskell

will enable us to gather data needed for debugging its parallel performance. It also means that the amount of information that we need to record is significantly smaller than if we want to do a full replay. In the next section, we discuss exactly what events we need to record in the logging phase.

We have built a prototype implementation of this modified execution replay in the Glasgow Haskell Compiler (GHC) and runtime system for Parallel Haskell [13][4]. Currently, logging of the events works by running a program under GHC with event logging support[5]. For replaying, we use the same compiled executable, with the `--replay` command line flag. This runs the simulator inside the GHC runtime system, which reads the events recorded in the logging phase and sequentially simulates the program execution. In Section 4 we provide additional details.

3.1 Events Needed for Replay

Execution replay relies on the amount of recorded information in the logging phase being tractable. Usually, most of the program execution consists in running code with a deterministic runtime behaviour, which can be replayed just by rerunning it. With the introduction of mutation, parallelism and non-deterministic data sources (e.g. random numbers, I/O, signals), the execution path (and, hence, the ordering of certain events in the program execution) can change. If the program execution is to be reproduced, all the events that introduce non-determinism in the program execution need to be recorded. In imperative languages, the biggest problem is to track the mutation of data, which may be

[4] Its development can be followed on http://github.com/hferreiro/ghc.

[5] Using the flag `-eventlog` to compile and the runtime system flag `-ls` when executing the program.

shared between different threads at any time, and this may require every access to shared memory to be logged. Current mechanisms for doing this efficiently rely on very elaborated protocols of page ownership tracking at the operating system level [7].

In a pure and lazy functional language, on the other hand, the situation is much simpler. The interactions between threads (which are the main reason for the existence of different execution paths in parallel programs) are greatly simplified. Data dependencies between threads are handled transparently, without the use of locks and other synchronising mechanisms. Once a computation has been evaluated to normal form, then the runtime system enforces read-only access. Furthermore, due to laziness, any unevaluated data will be updated by at most one thread.

For simplicity, we consider only pure computations (i.e. those that do not use any side-effects, such as I/O and concurrent data mutation). Also, as mentioned earlier, we are focusing on parallel profiling, which means that we are only interested in each thread's progress and the coordination between threads. Given this, for Parallel Haskell programs there is only a small number of events that needs to be recorded:

- thread interactions: *thread run*, *thread stop*, *thread block* (when a thread is blocked on some data being evaluated by some other thread);
- scheduling events: *thread migrate* (when threads are migrated between cores), *new spark* (for creation of parallelism), *steal spark* (load balancing event), *run spark* (when a parallel expression is picked by its owner thread);
- task related: *acquire capability*, *release capability* (to track ownership of capabilities).

Besides these, there are also some additional events very specific to the internal details of the GHC runtime system. In Section 4, we give more details about the implementation of the recording of events in GHC.

3.2 Usage of Execution Replay

The key observation for our work is that we are *simulating* the previous execution of the same program, rather than rerunning it. The replay is deterministic in its runtime behaviour, and only depends on the events that were recorded in the original run of the program. This makes it ideal for performance debugging of functional programs, since gathering more profiling data does not have any impact on the ordering of the events in the replay. It might only increase the time the replay may take, which is not of great importance in debugging.

Our ultimate goal is to integrate execution replay with the ThreadScope visualisation tool. In that way, we would have a GUI tool that would enable us to pause the replay at the points where the parallel performance starts to degrade, and then turn on the appropriate kind of profiling that would enable us to get a better insight into the problems encountered. In Section 5, we show a worked example of using execution replay to debug a non-trivial parallel program (quicksort). We now discuss a few hypothetical use cases of such a tool:

– For parallel programs that perform badly due to a large amount of unevaluated data shared between threads, which is reflected in frequent blocking of threads, we can replay the program execution without any profiling data up to the point where blocking starts to occur, then turn on profiling to investigate what data are the threads blocking on. At this point, we could take advantage of cost centre profiling to link the heap data to the expressions in the source code to which they relate, so that we can find out where exactly in the program source do these data hotspots come from. We may then rewrite the original program to avoid sharing at these particular points.
– In large parallel programs, we might be interested in different profiling data during different stages of the execution. In some stages, we might be only interested in granularity of the threads created from sparks, in others, we might be interested in discovering what data is being garbage collected. The possibility of dynamically adjusting the type and level of profiling detail during replay is one of the main motivations for using execution replay.
– For some parallel programs, there may be very subtle bugs which produce one bad execution out of many. It is not very useful to have to rerun your program many times until you reproduce a pathological behaviour. By using execution replay, the only requirement is to have a trace of the target execution. Then, it can be replayed as many times as needed with the confidence that the same wrong behaviour is being analysed.

4 GHC Implementation Details

Although the result of a Parallel Haskell program is deterministic, its runtime behaviour might not be. In this section, we describe some of the internals of the GHC runtime system, focusing on the parts that contribute to the non-deterministic behaviour of application execution[6]. We then describe in more detail how we implemented the logging and replaying phases of execution replay in GHC.

4.1 The GHC Runtime System

The Glasgow Haskell Compiler is a state-of-the-art compiler and parallel runtime system for the pure lazy functional language Haskell [10]. It achieves great flexibility by using a lightweight thread model, where multiple logical Haskell threads are mapped into one single OS thread which runs concurrently with others (see Figure 3). The whole runtime system is organised in three layers of abstraction: capabilities, tasks and threads. A *capability* is a virtual core in which Haskell code is run. Each time a new thread is created at the Haskell level, it will be appended to the run queue of its capability. To run the code of these threads, real OS threads are needed. This is the mission of *tasks*: each task corresponds to an OS thread which tries to become the owner of a capability. Once a capability has been acquired, the task will run a scheduler cycling through the capability's run queue and assigning a time slice to each Haskell thread.

[6] A much more complete description of GHC can be found in [11].

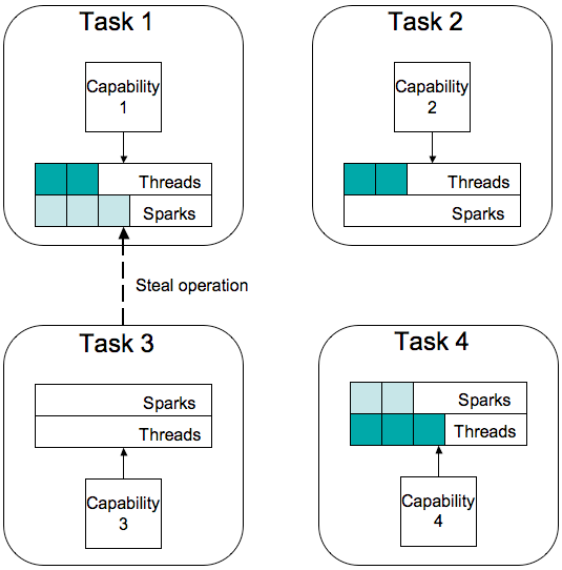

Fig. 3. Overview of the GHC Runtime System

Besides finishing its time slice there are other mechanisms by which a thread can lose control of the CPU: blocking on the evaluation of a shared value, a stack or heap overflow, or exceptions. Additionally, threads can migrate to idle capabilities to increase parallelism.

The basic primitives for parallel programming are *par a b* and *pseq a b*[7]. *par* denotes that it would be useful to evaluate its first parameter in parallel to the second which is returned as result [15]. *pseq* makes it possible to order the evaluation of two expressions by ensuring that its first parameter is evaluated to weak head normal form before returning its second parameter [11].

When a thread evaluates the expression p `` `par` `` q, a *spark* for p is created, and the thread continues with the evaluation of q. Sparks are just pointers to the part of the graph that represents the source expression. They are kept in *spark pools*, with one spark pool per capability. Each spark is eventually converted into a thread, or discarded if the expression it points to is already under evaluation, or not needed at all. Load balancing across capabilities is done using work stealing and pushing, where idle capabilities steal sparks, and threads are pushed from busy ones.

[7] Not to be confused with *seq* which is strict in both of its arguments but does not enforce an ordering in its evaluation [11].

4.2 Logging Phase

From the discussion above, we identified the set of events related to the possible non-determinism in the execution of parallel programs under GHC that we need to record. A mechanism for event logging already exists in GHC [6], and it supports logging of some basic events that are used for visualisation with ThreadScope. We have significantly enhanced the logging mechanism, adding several new events and changing some of the existing ones. We can group the needed events by the parts of the runtime system they are related to:

Thread scheduling. Logical Haskell threads executed on the same capability are scheduled in a round-robin fashion. Each thread runs in a capability until one of three things occur:

- the thread runs out of heap or stack space (in the first case garbage collection needs to be performed before any thread can continue evaluation),
- the thread blocks on some expression being evaluated, or
- the predefined time slice expires.

In all of the above cases, the thread is preempted and the next thread in the queue is selected for evaluation. If we want to replay how threads are interleaved, we need to be able to tell how much evaluation a thread has done in a given time slice. This amount of work changes in different executions because of how modern CPUs work. Thankfully, GHC preempts threads only when they make a heap check. The already existing *thread run* event already provides us with the data to identify which thread began running in a capability. We then modified the *thread stop* event to additionally store the amount of allocation the thread did in its time slice. For the case in which a thread blocks on an expression being evaluated by another thread, we enhanced the *thread block* event by adding information about the threads involved. A *thread wakeup* event is recorded when its execution can be resumed.

Load balancing. As a consequence of the previous discussion, the number of sparks created in the same time period in different program executions can be different, and also the times at which capabilities become idle (and, therefore, the need to perform spark stealing or thread pushing) can be different. This means that we need to record the events related to spark creation and migration, and also events related to threads being pushed to capabilities. We, therefore, need *new spark* (that occurs when a new spark is created), *spark steal* (that occurs when a capability steals a spark) and *thread migrate* (that occurs when a thread is migrated between capabilities) events to be logged. For these events, we need to record exactly which capability became idle, which spark it stole from which capability, or which thread was pushed to it.

Capability ownership. The task-related events that were described before, *acquire capability* and *release capability*, are new events we added so that we could track which task was responsible for the execution of threads in a capability.

An excerpt from the trace of the *simplePar* application described in Section 2 is shown below:

```
...
4177926000: cap 1: stopping thread 4 (stack overflow) (96 words allocated)
4177940000: cap 1: running thread 4
4180949000: cap 1: stopping thread 4 (heap overflow) (65024 words
 / allocated)
4180979000: cap 0: stopping thread 3 (blocked on blackhole owned by
 / thread 4) (25253 words alloced)
4181027000: cap 0: task 1 releasing
4181146000: cap 1: running thread 4
...
```

4.3 Replay Phase

The replaying phase works by spawning an independent *scheduler thread* at the beginning of the program execution. This thread initialises the runtime system and makes sure that all tasks stop after being created. Then, in a loop, it reads the recorded events ordered by time. If the event is *thread run* or *thread wakeup*, the scheduler thread checks the capability responsible for the event, and allows the corresponding thread to progress until it is stopped (once it has done the same amount of work as in the recorded execution) or blocked. The rest of the events are needed to preserve the ordering between the threads that emit conflicting events (the same spark trying to be stolen by different threads, etc.). Respecting this ordering will allow the execution to be replayed without trouble.

5 Use Case: Why is Parallel Quicksort so Slow?

In order to show how execution replay can be used for performance debugging of non-trivial parallel programs, we come back to the quicksort example we presented in Section 2. Quicksort has gathered a lot of attention recently in teaching parallel functional programming at Universities [4], since it is an example of a program which "seems" rather trivial to parallelise, yet for which obtaining good speedups (especially using lazy languages) is quite challenging.

A high-level, integrated profiling tool, designed with the use cases detailed in Section 3.2 in mind, is still work in progress, so for this example we show how to use execution replay in conjuction with a custom low-level tool for annotating the source code.

We saw in Section 2 that the obvious method of parallelising this program does not work as expected. In order to come up with a better parallel program, we first made some optimisations to the sequential version. We implemented our own strict version of the *partition* function so that we could avoid the overhead of lazy evaluation caused by computing the sublists on demand. Next, we got rid of the append operator ++, which requires multiple traversals of the same lists when it is applied left-recursively, as in our case. For this, we used an accumulator in

which the resulting list is being constructed. First, we start with the whole list to be sorted and an empty accumulator. Then, at each recursive step, the pivot is accumulated into the sorted higher sublist. When there are no more elements to sort, the accumulator is returned as the fully sorted list.

```
qsort :: [Int] → [Int]
qsort xs = seqSort xs []
  where seqSort []       zs = zs
        seqSort (x : xs) zs = seqSort lo (x : seqSort hi zs)
          where (lo, hi) = partition x xs
partition :: Int → [Int] → ([Int], [Int])
partition x xs = go xs [] []
  where go []       ts fs = (ts, fs)
        go (y : ys) ts fs
          | y < x       = go ys (y : ts) fs
          | otherwise   = go ys ts       (y : fs)
```

Similarly to the first time, we tried to naively parallelise this code in the same way as we did in Section 2. Given the changes mentioned above, we expected to avoid the sequential phase that occurs at the end of the execution.

```
psort1 n xs = go n xs []
  where go _ []       zs = zs
        go n (x : xs) zs
          | n > 0       = r `par` go (n − 1) lo (x : r)
          | otherwise   = seqSort (x : xs) zs
          where r = force (go (n − 1) hi zs)
                (lo, hi) = partition x xs
```

We measured the speedups of this program on a machine with two Intel Xeon 2.93GHz CPUs, each of them having four cores. Each CPU owned 8MB of L2 cache, that was shared between all of its cores. The total amount of RAM was 64GB. In the speedups figures below, we took the mean time over five runs of each program with the same input, a list consisting of 10 million elements.

Figure 4 shows the speedups against the sequential version of the algorithm, measured by using from two up to eight cores. From the figure, we can observe that we are actually getting significant slowdowns as we use more cores.

No. cores	Speedup
2	1.49
4	0.78
8	0.60

Fig. 4. Speedups of *psort1* with an input list of 10 million elements

In order to debug the performance of this program, we used again ThreadScope to get an overview of the thread activity. Figure 5 shows the output from ThreadScope after running our program using two cores.

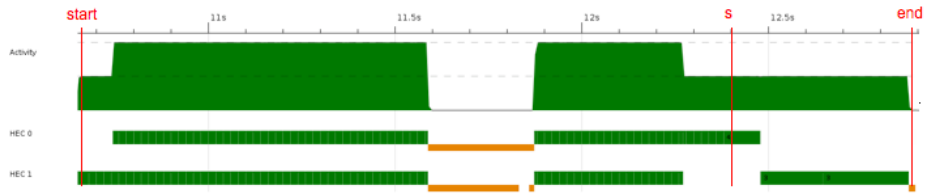

Fig. 5. ThreadScope profile of *psort1*

While garbage collection pauses were mostly fixed, we can see that we still have the same serialisation problem we had in the initial parallel version in Section 2, and that getting rid of ++ operator did not help at all.

We now used execution replay to discover which part of the program is responsible for the sequential phase at the end of the execution. We developed some custom tools on top of our system to be able to register timestamps when the evaluation of an annotated expression is finished and to analyse the output produced. By using execution replay, we were sure that the same execution was reproduced and so, the output data matched the original ThreadScope profile. To focus on the interesting parts of the program, we added two checkpoints: `start`, which is the point after reading the input list, and `end` which marks the end of the program (see Figure 5). We then replayed the program and processed its output to obtain the following report:

```
   188.020 (   93.914) cap 0: partition [10.837.539]
 1.240.278 (1.052.258) cap 0:   seqSort [11.889.797]
 1.747.763 (  507.485) cap 0:   seqSort [12.397.282]
 1.747.766 (        3) cap 0:     force [12.397.285]
 1.828.627 (   80.861) cap 0:     force [12.478.146]

         0 (        0) cap 1:     start [10.649.519]
    94.106 (   94.106) cap 1: partition [10.743.625]
   170.970 (   76.864) cap 1: partition [10.820.489]
   732.638 (  561.668) cap 1:   seqSort [11.382.157]
 1.621.573 (  888.935) cap 1:   seqSort [12.271.092]
 1.996.394 (  374.821) cap 1:     force [12.645.913]
 2.225.849 (  229.455) cap 1:     force [12.875.368]
 2.225.852 (        3) cap 1:       end [12.875.371]
```

Each line shows the timestamps for the completion of each annotated function in the program. First, the relative time against `start` is presented. Next, the rel-

ative time against the previous function timestamp and the absolute timestamp are shown in brackets. Each event is classified according to its capability.

The relevant aspect of this data is that the sorting process has finished by the time the sequential phase begins. We can see this because the timestamp of the finish time of the last call to the *seqSort* function on capability 0 is 12.397s (checkpoint `s` in Figure 5) and, from the ThreadScope profile, we can observe that the sequential phase starts at a timestamp around 12.3s. After the checkpoint `s`, only the timestamps of the *force* functions are left. So, the sequential phase at the end must correspond to the execution of these functions.

The conclusion is that the program execution is almost perfectly balanced between the two cores while the parallel threads are sorting their parts of the list (the timestamps for the completions of the calls to *seqSort* are similar in each capability). But then, because of the *force* call, each thread needs to traverse the sublist passed in the accumulator *zs*. This sublist is being sorted by another thread, so the thread evaluating the *force* call gets blocked immediately, waiting until *zs* has been evaluated. Only then can it finish traversing it. When finished, this thread returns the sorted list and allows its parent thread to also finish evaluating its *force* call. This linear process gets worse as more threads are involved in it. This is the reason why the speedups get worse as we add more cores.

In the end, the same behaviour we tried to prevent by avoiding the $+\!\!+$ operator, i.e. sequential traversal of the sorted list, is reproduced by evaluating to normal form each of the sublists.

This analysis suggests that the way to fix this behaviour is to replace the function *force* with a function that would immediately return when the tail of the list being forced is already in normal form. To this end, we implemented a custom version of quicksort which operates on a datatype $List\ a$ (instead of a regular list) as its input. This new type has the same $Nil/[\,]$ and $Cons/:$ constructors as regular lists, and also an additional constructor *Done*. The *Done* constructor has a list of elements as argument, and is used to mark this list as fully evaluated. Together with this new type, we introduced a $toList :: List\ a \rightarrow [a]$ function which takes a $List\ a$ as input and returns its corresponding regular list in normal form. Its behaviour is similar to our usage of *force*, with the exception that it terminates if a $Done\ xs$ element is found:

$$
\begin{aligned}
&\textbf{data}\ List\ a = Nil \mid \quad Cons\ a\ (List\ a) \mid Done\ [a] \\
&toList :: List\ a \rightarrow [a] \\
&toList\ Nil \qquad\quad = [\,] \\
&toList\ (Cons\ x\ xs) = \textbf{let}\ xs' = toList\ xs \\
&\qquad\qquad\qquad\quad\ \ \textbf{in}\ x\ `seq`\ xs'\ `seq`\ x : xs' \\
&toList\ (Done\ xs) \quad = xs
\end{aligned}
$$

Now, by making use of the former definitions, we can implement a version of *psort1* in which the threads evaluating the higher half of the list, *hi*, will mark it as already evaluated, so that the ones sorting the other half will find a $Done\ xs$ value and directly return *xs* instead of traversing it again:

```
psort2 :: Int → [Int] → [Int]
psort2 n xs = toList (go n xs Nil)
  where go _ [] zs = zs
        go n (x : xs) zs
          | n > 0 = r `par` go (n − 1) lo (Cons x r)
          | otherwise = seqSort (x : xs) zs
          where r = Done $! toList (go (n − 1) hi zs)
                (lo, hi) = partition x xs
```

The speedups for *psort2* are shown in Figure 6. We can observe much better speedups that for *psort1*. For two cores, the speedup is almost linear. However, it gets worse when using more cores. This decrease in the performance can be attributed to the fact that each thread is created only after the list has been partitioned. The same thing will happen to the next threads once the generated sublist are partitioned again. So, if we need to use more threads, it will take longer to create them, increasing the initial sequential phase.

No. cores	Speedup
2	1.90
4	2.35
8	2.75

Fig. 6. Speedups of *psort2* with an input list of 10 million elements

6 Related Work

Previous approaches to performance profiling of Parallel Haskell programs involve the use of simulators such as GranSim [9] or parallel cost centre profiling [3]. GranSim was developed as an instrumentation of the GHC runtime system that allowed the programmer to gather statistics of the program which was simulated to run in a distributed machine with a customizable environment (e.g. network delay). Events could be visualised in a similar way to ThreadScope. The same event log format was used by the parallel profiler for the GUM parallel implementation [16]. Similar techniques are used by the more recent ThreadScope [6] and EdenTV [1] visualisers.

Our approach enhances profiling by using a kind of simulated environment, which, in contrast to GranSim, does not emulate any real hardware but replays a previous run. This technique is known as *execution replay* [14]. So far, it has been used almost exclusively for debugging instead of performance profiling. Most execution replay systems allow any program to be replayed without recompilation [7]. The most difficult problem these systems have to solve is that of shared memory interactions, something we can completely ignore because our

source code is purely functional. In addition, some of these systems also try to replay the scheduling of threads (a requirement in our case), but they do so by using hardware counters [5, 8], which makes them hardware dependent and subject to inaccurate measurements [12, 17]. Another benefit of our approach is that we can modify the original program at will, as long as it produces the same allocations, in order to gather more information, and the replay will still be valid.

7 Conclusions and Future Work

In this paper, we described a prototype implementation of the execution replay mechanism in the GHC compiler for Parallel Haskell. We also described how this mechanism can be used to obtain a better insight of the parallel behaviour of functional programs, which makes it very useful for performance debugging of such programs. We have presented a use case of execution replay for parallel debugging, using a parallelisation of the well-known quicksort algorithm as an example. Whe showed that, despite quicksort being a program which seems easy to parallelise, it contains a number of hidden caveats that make obtaining good speedups quite challenging. Hence, being able to obtain better profiling information is vital in order to understand its behaviour and discover the bottlenecks.

This paper presents the first implementation of the execution replay mechanism in the context of a lazy functional language. In addition, this is the first time execution replay is used for performance debugging. Our focus on pure functional languages and on parallel performance debugging significantly relaxed the assumptions that we need to make about the replay. We are not restricted to having to reproduce *exactly the same execution* as the original one. The replayed execution can differ from the original one, as long as they both have the same parallel behaviour. This significantly reduces the amount of logging information that is required for replay, making it much less expensive that when used in imperative languages for replaying the exact state of the program at each point of its execution. It also allows dynamic enabling and disabling of data gathering modules during the replay.

With execution replay as a foundation, we are able to build better profiling tools which will allow functional programmers to better understand and fix many parallel programs for which there were no tools to deal with. In the future, we plan to implement these tools by integrating already existing profiling and visualisation approaches (such as ThreadScope and cost centre profiling) with execution replay. We are also in the process of extending execution replay for programs with side-effects. Finally, we plan to demonstrate the effectiveness of replay-driven performance debugging on a larger set of parallel programs.

References

1. Berthold, J., Loogen, R.: Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In: Parallel Computing: Architectures, Algorithms

and Applications. Advances in Parallel Computing, vol. 15, pp. 121–128. IOS Press (Feb 2008)
2. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., De Bosschere, K.: A Taxonomy of Execution Replay Systems. In: Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (2003)
3. Hammond, K., Loidl, H.W., Trinder, P.: Parallel Cost Centre Profiling. In: Proceedings of the Glasgow Workshop on Functional Programming. Ullapool, Scotland (Sep 1997)
4. Hughes, J., Sheeran, M.: Teaching Parallel Functional Programming at Chalmers. In: Draft Proceedings of the 1st International Workshop on Trends in Functional Programming in Education (Jun 2012)
5. Itskova, E.: Echo: A deterministic record/replay framework for debugging multithreaded applications. Master's thesis, Imperial College, London (Jun 2006)
6. Jones Jr., D., Marlow, S., Singh, S.: Parallel Performance Tuning for Haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. pp. 81–92. Haskell'09, ACM (2009)
7. Laadan, O., Viennot, N., Nieh, J.: Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 155–166. SIGMETRICS'10, ACM (Jun 2010)
8. Lee, D., Said, M., Narayanasamy, S., Yang, Z., Pereira, C.: Offline symbolic analysis for multi-processor execution replay. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 564–575. MICRO 42, ACM (2009)
9. Loidl, H.W.: Granularity in Large-Scale Parallel Functional Programming. Ph.D. thesis, Department of Computing Science, University of Glasgow (Mar 1998)
10. Marlow, S.: Haskell 2010. Language Report (2010), http://www.haskell.org/onlinereport/haskell2010
11. Marlow, S., Peyton Jones, S., Singh, S.: Runtime Support for Multicore Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 65–78. ICFP'09, ACM (Aug 2009)
12. Mathur, W., Cook, J.: Toward Accurate Performance Evaluation using Hardware Counters. In: Proceedings of the Applications for a Changing World, ITEA Modeling & Simulation Workshop (Dec 2003)
13. Peyton Jones, S.L., Hall, C.V., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell compiler: a technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference (1993)
14. Ronsse, M., De Bosschere, K., Chassin de Kergommeaux, J.: Execution replay and debugging. arXiv:cs/0011006 (Nov 2000)
15. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.: Algorithms + Strategy = Parallelism. Journal of Functional Programming 8(1), 23–60 (Jan 1998)
16. Trinder, P.W., Hammond, K., Mattson Jr., J.S., Partridge, A.S., Peyton Jones, S.: GUM: A Portable Parallel Implementation of Haskell. In: Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation. pp. 79–88. PLDI, ACM (May 1996)
17. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software. pp. 23–32. ISPASS, IEEE (Apr 2009)