PROFILING OF PARALLEL PROGRAMS IN A NON-STRICT FUNCTIONAL LANGUAGE

HENRIQUE FERREIRO

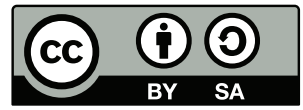
Supervised by Laura Castro and Kevin Hammond

Doctoral Thesis / 2015

Department of Computer Science

UNIVERSIDADE DA CORUÑA

Henrique Ferreiro: *Profiling of parallel programs in a non-strict functional language*, 2015.



Ao meu pai e ao meu fillo, que,
cada un á súa maneira,
souberon esperar até que rematase a tese.

ACKNOWLEDGMENTS

First of all, I thank my former supervisor Víctor Gulías for his guidance at the start of my research career. He always provided support while under his supervision and backed my interest in functional programming and Haskell.

I am very grateful to Laura Castro for accepting to supervise my thesis and for the time devoted to this task, specially in the last few months. After all this time working together, I consider her more of a friend than my advisor.

Kevin Hammond was the reason I started working with GpH and eventually focused my thesis on its final topic. The many discussions we held at his office kept me on the right track to finish the task at hand.

I want to thank all my lab colleagues for making coming to work a real pleasure. Specially, I need to mention Castro and Macías, with whom I had many coffees and conversations; and Santi, who reviewed a draft of this thesis. Chris and Vladimir, who I met during my stays at St Andrews, became good friends and helped me a lot in my research.

I thank my family for their encouragement through these years and their patience. As to my partner, Alba, finishing this thesis meant spending many hours away from her at a really important time. Without her support, I would have not finished it.

Lastly, I thank Aldán for waiting to be born until a few days after the thesis was finished.

ABSTRACT

Purely functional programming languages offer many benefits to parallel programming. The absence of side effects and the provision for higher-level abstractions eases the programming effort. In particular, non-strict functional languages allow further separation of concerns and provide more parallel facilities in the form of semi-implicit parallelism. On the other hand, because the low-level details of the execution are hidden, usually in a runtime system, the process of debugging the performance of parallel applications becomes harder. Currently available parallel profiling tools allow programmers to obtain some information about the execution; however, this information is usually not detailed enough 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 thesis, we design and implement a parallel profiling framework based on *execution replay*. This debugging technique makes it possible to simulate recorded executions of a program, ensuring that their behaviour remains unchanged. The novelty of our approach is to adapt this technique to the context of parallel profiling and to take advantage of the characteristics of non-strict purely functional semantics to guarantee minimal overhead in the recording process. Our work allows to build more powerful profiling tools that do not affect the parallel behaviour of the program in a meaningful way. We demonstrate our claims through a series of benchmarks and the study of two use cases.

RESUMO

As linguaxes de programación funcional puras ofrecen moitos beneficios para a programación paralela. A ausencia de efectos secundarios e

as abstraccións de alto nivel proporcionadas facilitan o esforzo de programación. En particular, as linguaxes de programación non estritas permiten unha maior separación de conceptos e proporcionan máis capacidades de paralelismo na forma de paralelismo semi-implícito. Por outra parte, debido a que os detalles de baixo nivel da execución están ocultos, xeralmente nun sistema de execución, o proceso de depuración do rendemento de aplicacións paralelas é máis difícil. As ferramentas de *profiling* dispoñibles hoxe en día permiten aos programadores obter certa información acerca da execución; non obstante, esta información non acostuma a ser o suficientemente detallada para determinar de maneira precisa a causa dalgúns problemas de rendemento. A miúdo, isto débese a que o custe de obter esa información sería prohibitivo para unha execución completa do programa. Nesta tese, deseñamos e implementamos unha plataforma de *profiling* paralelo baseada en *execution replay*. Esta técnica de depuración fai que sexa posible simular execucións previamente rexistradas, asegurando que o seu comportamento se manteña sen cambios. A novidade do noso enfoque é adaptar esta técnica para o contexto do *profiling* paralelo e aproveitar as características da semántica das linguaxes de programación funcional non estritas e puras para garantizar unha sobrecarga mínima na recolección das trazas de execución. O noso traballo permite construír ferramentas de *profiling* máis potentes que non afectan ao comportamento paralelo do programa de maneira significativa. Demostramos as nosas afirmacións nunha serie de *benchmarks* e no estudo de dous casos de uso.

RESUMEN

Los lenguajes de programación funcional puros ofrecen muchos beneficios para la programación paralela. La ausencia de efectos secundarios y las abstracciones de alto nivel proporcionadas facilitan el esfuerzo de programación. En particular, los lenguajes de programación no estrictos permiten una mayor separación de conceptos y proporcionan más capacidades de paralelismo en la forma de paralelismo semi-implícito. Por otra parte, debido a que los detalles de bajo nivel de la ejecución

están ocultos, generalmente en un sistema de ejecución, el proceso de depuración del rendimiento de aplicaciones paralelas es más difícil. Las herramientas de *profiling* disponibles hoy en día permiten a los programadores obtener cierta información acerca de la ejecución; sin embargo, esta información no suele ser lo suficientemente detallada para determinar de manera precisa la causa de algunos problemas de rendimiento. A menudo, esto se debe a que el costo de obtener esa información sería prohibitivo para una ejecución completa del programa. En esta tesis, diseñamos e implementamos una plataforma de *profiling* paralelo baseada en *execution replay*. Esta técnica de depuración hace que sea posible simular ejecuciones previamente registradas, asegurando que su comportamiento se mantiene sin cambios. La novedad de nuestro enfoque es adaptar esta técnica para el contexto del *profiling* paralelo y aprovechar las características de la semántica de los lenguajes de programación funcional no estrictos y puros para garantizar una sobrecarga mínima en la recolección de las trazas de ejecución. Nuestro trabajo permite construir herramientas de *profiling* más potentes que no afectan el comportamiento paralelo del programa de manera significativa. Demostramos nuestras afirmaciones en una serie de *benchmarks* y en el estudio de dos casos de uso.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

CAF	Constant applicative form
EDT	Evaluation dependence tree
ER	Execution replay
FP	Functional programming
GC	Garbage collection
GHC	Glasgow Haskell Compiler
GpH	Glasgow parallel Haskell
Hp	Heap pointer
HpLim	Heap pointer limit
LLVM	Low-Level Virtual Machine
LML	Lazy ML
OS	Operating system
RTS	Runtime system
STG	Spineless Tagless G-machine
TCO	Tail call optimisation
WHNF	Weak head normal form

1 INTRODUCTION

This chapter presents the topic of parallel programming and profiling in the context of a functional language. We give a definition of parallel programming and give a brief overview of parallel programming models, and profiling and debugging tools in section 1.1. We also show the benefits of functional programming to the parallel paradigm and some of the shortcomings of currently available tools for profiling. Then, in section 1.2, we outline the solution we have developed to overcome some of the aforementioned problems. In section 1.3, we list the contributions made by this thesis.

1.1 THE CHALLENGES OF PARALLELISM

Since the beginning of computing history, huge amounts of research have been spent in the development of technology that enables computations to be calculated in parallel as a way of improving execution runtime. The limits on single-core processor scalability have recently put pressure on the ability to take advantage of parallel hardware. The duplication of computing power every two years derived from the increase in transistors per chip, known as Moore's law, stalled several years ago when miniaturisation started to find physical limits such as heat dissipation, increasing power consumption and current leakage [1]. Even though Moore's law still holds, the increase in transistors per chip is accomplished using multicore architectures. What was once a matter of waiting some time to acquire a faster processor, now requires a fundamental change in the way programmers write computer programs to take advantage of concurrent execution.

Today's desktops and laptops are usually provided with single-chip multicores, but current research is already bringing manycore hardware

and heterogeneous platforms where the CPU is accompanied by a highly-parallel GPU [2]. Improved parallelism in hardware challenges current programming models to keep up with the increase in concurrency. Even though parallel programming has been possible since long ago, making it easy to use and, at the same time, provide the mechanisms and tools to understand and improve its performance and scalability is still an open research problem.

What makes parallel programming hard is that, in order to obtain speedups against a sequential implementation, a parallel program needs to decompose its computations into independent *tasks* (units of work) that perform their work in parallel with each other. It follows then that the programmer needs the tools and ability to identify these independent tasks in a way that the total work share of the program is as evenly distributed as possible. There is complexity involved in doing this, such as figuring out ordering dependencies or finding independent work. Additionally, it is quite possible that there needs to be some form of coordination between the multiple tasks being run because they usually need to join their results, or their inputs come from a shared data source. In these cases, undesirable situations like deadlocks or starvation need to be avoided. This complexity is the reason code parallelisation has not been completely automated yet, and the motivator behind the multiple libraries and tools that are created to try to ease the programming and to profile the performance of parallel programs.

As a summary, in order to increase performance, programming languages and runtimes need to expose parallel primitives and/or libraries to programmers, so that computer programs can make the most of current highly-parallel hardware. This style of programming involves more challenges than sequential programming and the tasks of both debugging and profiling this programs is a big part of those.

1.1.1 *Parallel programming models*

As with other programming paradigms, parallel programming has evolved from an unstructured approach to the development of multiple parallel

programming models, intended to solve the challenges described above. A *parallel programming model* can be defined as a set abstractions and tools that allows to write parallel programs and provide an implementation so that the resulting program can be executed in a variety of parallel hardware. They usually consist in parallelising compilers, parallel languages or libraries.

Parallel programming models can be generally classified using two orthogonal properties: process interaction and problem decomposition. The first property establishes how independent tasks interact with each other and is closely related to the hardware categories. We can find here shared memory, message passing and implicit parallelism models. As for problem decomposition, it is related to the parallel structure of the program and divides parallel problems in task-parallel and data-parallel problems.

We provide a brief description of some of these models:

SHARED MEMORY: in this model, tasks share a common address space in which to write and read data. To solve the problem of concurrent access to shared resources, synchronisation mechanisms are available such as locks, semaphores or monitors. Independent tasks can be created by using processes or spawning threads. In general, this approach is very low-level and requires a lot of skill to avoid data races and deadlocks. There are some approaches that make it easier by relying on the compiler to generate the parallel code. For example, OpenMP [3] is an industry standard which provides compiler directives for a number of programming languages allowing the programmer to parallelise loops and protect global data.

MESSAGE PASSING: this model is based around explicit communication, where data that is needed by different tasks is sent and received by passing messages. It avoids the need to synchronise access by forcing the programmer to explicitly design a communication protocol between tasks and sharing nothing. As a drawback, it may incur in additional memory usage because of data being copied when sent. Popular implementations of this model

are the MPI [4] and PVM [5] libraries and the Actor model [6] or π-calculus [7], mathematical formulations of the model.

DATA PARALLELISM: this model is an instance of a single-instruction multiple-data architecture where tasks execute the same code in independent chunks of some data. This model can work in a distributed memory architecture by sending the chunks to their corresponding nodes and then joining the results back. It is the models used to program GPUs. Unified Parallel C [8] or CUDA [9] are some implementations of this model.

TASK PARALLELISM: this model easily defined in comparison to data parallelism. Instead of splitting the data to perform parallel computations in each piece, independent tasks are identified to execute in parallel using the same or different data. It is a very broad parallel programming model and most of the libraries referenced earlier can be used for task parallelism.

The taxonomy presented here can serve to classify the particular way in which a parallel problem is solved but, in general, parallel implementations fall under more than one category, providing libraries and/or tools to combine multiple models.

1.1.2 *Parallel functional programming*

While some parallel programming models can help to structure a parallel program, most of the work comes from the understanding of the program and the ability to reorganise and decompose it (or the data it needs to manipulate) into smaller tasks with little dependencies between each other. Many of the difficulties in doing this are related to the underlying language technology. For the most popular paradigms, imperative and object-oriented programming, *side effects* are an integral part of their programming model. Specifically, *data mutability* is a pervasive feature that makes it very hard to reason about data dependencies and ordering, fundamental to achieve an effective parallel structure. This is especially

important in the immediate future, where the need to scale to thousands of nodes will only make the problem more prominent.

Even though the functional programming (FP) paradigm has existed for as long as imperative programming, it has only been in recent times that it has started to become mainstream. Modern programming languages usually employ a number of features from different programming paradigms, making it hard to delimit what makes a language functional or imperative. Nevertheless, a functional language can be defined as a programming language in which the main building-block is the function definition, and complex programs are built by composing smaller functions together [10]. However, when one thinks of a FP language, there are a whole range of high-level abstractions that are commonly associated with it:

HIGHER-ORDER FUNCTIONS: these are functions that allow passing other functions as arguments and/or return functions as their results. This feature is also described as "functions as first-class values". Higher-order functions serve as glue to compose functions generically and can be emulated in imperative languages by function pointers or classes.

RECURSION: this feature allows a function to call itself as part of its own execution, possibly with different parameters. This serves as the alternative to loops in imperative languages. Using higher-order functions, many recursive patterns can be abstracted away resulting in more code reuse.

PURITY: a pure function is a function that has no side effects when it is evaluated. As a consequence, what are usually called *variables* in imperative languages are just names used to identify constant values and, given the same inputs, the result of a pure function is always the same. The absence of side effects enables many program optimisations (e.g. memoisation, discard unused expressions, etc.) and prevents many bugs that arise from state mutation.

REFERENTIAL TRANSPARENCY: this feature is closely related to purity. Referential transparency means that any expression can be in-

terchanged with its definition without changing the meaning of the program. This property requires that evaluating an expression always result in the same output, but it is not as strong as purity. Referential transparency enables *equational reasoning*, making it easier to prove some properties about the code, or to apply program transformations.

EXPRESSIVE TYPE SYSTEM: many functional languages are equipped with type systems more expressive than those of mainstream programming languages that can provide many correctness guarantees without the inconvenience of writing cumbersome type signatures by using type inference. The use of algebraic data types and pattern matching makes FP suitable to express and manipulate complex data structures.

Laziness and Haskell

There is one characteristic of functional languages we have omitted up to this point: *strictness*. Strictness defines how a program is evaluated: either a function evaluates its arguments before evaluating its body (*eager* or *strict evaluation*) or it does so the other way around (*non-strict* evaluation), so that arguments are only evaluated on demand, when they are needed.

In this thesis we use a particular functional language with non-strict semantics called Haskell [11]. The initial Haskell 98 language standard, edited by Peyton Jones and Hughes [12], was published in 1999. Even though the report defines Haskell as a non-strict functional language, it is common to use the term *lazy* because most of its implementations use a *call-by-need* evaluation strategy, implemented using graph reduction. That means that, not only the evaluation of expressions is postponed, but their results are shared. Because laziness is an implementation detail of non-strict semantics, and this convention is widespread, we use the terms interchangeably even if it is not entirely accurate.

There are many benefits associated with laziness, but some of the most useful are the following [13]:

- The ability to define new control structures. While other languages need to provide them built-in, a lazy language allows the user to define control structures as normal functions. For example, Haskell provides the following function:

  ```
  when :: (Monad m) => Bool -> m () -> m ()
  ```

  equivalent to the *then* branch of a conditional statement. Using strict semantics, a language would need either a macro system that did textual substitution or built-in support, as usually happens.

- The improvement of program *modularity* through separation of concerns. Laziness allows to define data producers independently of its consumers without having to worry about performance or correctness. Again, we provide the code for a common Haskell function to verify if a property holds in any element of a list:

  ```
  any :: (a -> Bool) -> [a] -> Bool
  any p = or . map p
  ```

  This kind of function reuse is impossible in a strict language, which should resort to use built-in operators with short-circuit evaluation such as `||` in C or `orelse` in Erlang to avoid computing the whole list after finding an element which fulfils the property `p`.

From now on, even while trying to remain general, we refer to a lazy purely functional language when talking about functional programming. In particular, we use Haskell in all code snippets. Additionally, Glasgow parallel Haskell (GpH) [14] is used as the chosen Parallel Haskell language extension.

Parallel Haskell

GpH introduces two language primitives to program parallel applications:

```
par :: a -> b -> b

pseq :: a -> b -> b
```

These are the basic primitives on top of which many parallel programming models can be implemented [15]. `par`, as defined by its type, evaluates to its second parameter. It introduces parallelism by *denoting* that it would be useful to evaluate its first parameter in parallel. The exact details of how this is done are defined by the underlying Haskell implementation. This mechanism is very flexible because the expression which is possibly being parallelised is treated as any other Haskell expression and, as such, its result is shared by any thread needing it, and also garbage collected. This allows to parallelise expressions with different degrees of granularity and use *speculative parallelism*, relying in the runtime system for both ensuring non-duplicate evaluation and discarding unused values. It is important to realise that lazy evaluation is needed so that the evaluation of `par a b` returns `b` without forcing the evaluation of `a`. `pseq` complements `par` by enforcing the evaluation ordering of two expressions. It is mainly used to make sure that the thread that evaluates an expression using `par` does not immediately evaluate the parallelised subexpression.

To illustrate how to introduce parallelism in an existing program using these two primitives, we provide the code for a parallel Fibonacci implementation in listing 1:

Listing 1: Parallel Fibonacci function

```
1 pfib :: Int -> Int
2 pfib 0 = 1
3 pfib 1 = 1
4 pfib n = n1 `par` n2 `pseq` n1 + n2
5   where
6     n1 = pfib (n-1)
7     n2 = pfib (n-2)
```

This common parallelisation of the Fibonacci function consists in evaluating the first call to calculate the previous Fibonacci number, `n1`, in parallel with the second one, `n2`. In this case, `pseq` forces the thread evaluating `pfib` to evaluate `n2` before `n1 + n2`. This allows `n1`, which has been "marked" for parallel evaluation, to be evaluated by a different thread.

Lazy evaluation and other functional programming abstractions bring many benefits to parallel programming and, in particular, to the semi-implicit model provided by GpH. We follow Hammond and Michelson [10, pp. 1–7] to name a few examples:

EASE OF PARTITIONING. Because of the lack of side-effects, given any two expressions with no data dependencies, their order can be reversed or they may be evaluated in parallel. Purity makes it easier to reorganise the program and evaluate independent tasks in parallel. Serialisation is only forced by data dependencies and explicit control dependencies (`if` and `case` expressions), so that many dependencies are avoided by removing assignment.

SIMPLE COMMUNICATION MODEL. Evaluation of parallel shared expressions works as an implicit communication channel. A thread demanding a result being evaluated by another thread is queued until the result is available and woken up when ready.

ABSENCE OF DEADLOCK. Compared to the imperative paradigm, deadlocks cannot happen in a purely functional program. Implicit control dependencies as the ones introduced by assignment in an imperative setting must be protected by using some form of locking mechanism. Parallel functional programs, on the other hand, can rely on graph reduction to block on the evaluation of an expression and share its result or duplicate the work if a call-by-name strategy is used.

STRAIGHTFORWARD SEMANTIC DEBUGGING. Purely functional programs give the same results when executed in parallel as when executed sequentially. The lack of side effects means that the evaluation strategy is going to be independent of the result. Because the result is deterministic, depending on the parallel programming model, it may be possible to program and debug a sequential version, and later add parallel constructs without affecting the correctness of the result.

EASY EXPLOITATION OF PIPELINING AND OTHER PARALLEL CONTROL CONSTRUCTS. The style of programming that functional programming imposes leads to a program structure with a lot of similarities to algorithmic skeletons [16]. The use of function composition is analogous to *pipelining* and many recursive algorithms use lists in a way similar to a *task farm*.

For many of these cases, lazy evaluation can help to introduce more parallelism because data dependencies introduce a sequential dependency only when a value is required, and not just as a function parameter.

Given these properties, writing parallel programs in a purely functional language such as GpH, is misleadingly simple. In many cases, it consists in identifying key places in the source code where coarse-grained computations are evaluated, and instructing its runtime system to evaluate them in parallel.

The scenario depicted above seems ideal, especially when it is compared to many imperative parallel models where race conditions and deadlocks are real concerns one has to be aware of. On the other hand, these very same properties that make it easy to do parallel programming can be a challenge when one wants to improve an already parallel program, or debug the performance problems of an underperforming one. Specifically, the lack of explicit program flow makes it harder to enforce the best evaluation ordering when it is already known. Also, implicit parallelism may get in the way of implementing problems with a well-defined parallel structure. The implementation of `pfib` shown earlier is a good example of this limitations. A relatively inexperienced programmer can realise that a threshold is needed to obtain good performance in divide and conquer algorithms, but, even then, we do not obtain a linear speedup as one would hope in such a simple example. An explanation of this fact is shown in section 5.2.1.

1.1.3 *Profiling parallel functional programs*

Usually, in an imperative setting, after getting an initial version for a parallel program, the next step consists in using profiling tools to obtain runtime information that helps in understanding its parallel behaviour. This information may include function execution time, thread scheduling, lock contention, etc. The use of this kind of tools can be very helpful in that the programmer can identify and focus on the bigger cause of slowness and also understand the problems underlying its performance.

The problem with a lazy functional language in relation to parallel programming is the lack of tools supporting useful profiling. This is true even for sequential programs. The reason behind this circumstance is the fact that interleaving the evaluation of functions as they are demanded prevents from accurately measuring their running times, and makes it very difficult to map an expression being evaluated to specific points in time. For example, in a common producer/consumer expression such as `take 5 primes`, instead of calculating a list of prime numbers and then returning the first 5 values, lazy evaluation means that each one of the first 5 prime numbers is going to be computed as they are consumed. So, if those numbers are printed separately, `primes` would suspend its evaluation in between each print action.

Additionally, in a parallel setting, any technique that tries to solve this problem becomes more difficult to apply because the increase in the amount of information per unit of time makes the overhead of collecting it larger. Also, the nondeterminism inherent to parallelism means that the runtime behaviour of the program run under a profiler in consecutive executions may differ substantially to that of the original execution of the program (without any profiling).

Some of these shortcomings cannot be directly fixed and are instead resolved by approximating its result. For example, to calculate function execution times, a statistical approximation can be obtained by using cost centres [17]. Even if they do not give an exact measure of the program execution times, cost centres partition a program execution using the percentual usage of each expression and allow to prioritise which parts of the program to focus on.

Mapping source expressions to points in time and, more importantly, ordering thread interactions and progress, though harder to obtain, is more useful when trying to optimise a program for parallel execution. Gathering this information is vital to decide whether some tasks should be parallelised or not and its granularity. We can classify this problem as an instance of the more general technique of software tracing. *Software tracing* is a popular technique that consists in low overhead event logging to acquire valuable runtime information at the time those relevant events occur. Again, in this case, the execution model of a lazy functional language makes it very hard to make sense of a raw log of the execution call graph, where functions evaluation would appear intermixed with the evaluation of their arguments. Additionally, providing useful data may require costly inspection of the running program and a huge increase in the amount of runtime events, making its effect on the parallel execution unaffordable.

Up until now, work in this area has been scarce. An extension to the previous work on cost centres [18] was used to apply the same technique in a parallel setting, although, as in the sequential case, it requires some runtime system changes that have a serious impact on program performance. Other techniques worked by doing a program simulation single-threadedly and using specialised code to extract additional execution data so that the program could be optimised iteratively before deploying it to actual parallel machines [19].

One of the latest efforts involves a design around the usage of the Glasgow Haskell Compiler's tracing subsystem to provide a general view of runtime events over time [20]. Even if some of its information is useful, in our view, it is still insufficient to be considered a mature and complete parallel profiling tool.

1.2 TOWARDS A NEW PROFILING TECHNIQUE

We have identified software tracing as a technique to acquire profiling information which allows to debug parallel performance problems, although it has already been pointed out how the amount and kind of in-

formation that needs to be extracted together with the nondeterministic nature of the execution becomes an obstacle in a parallel environment.

We have identified an amount of runtime information that could provide valuable data in order to understand the behaviour of parallel programs: thread interactions in relation to source expressions (which expression has the most contention), wasted work (what parallel expressions were discarded), etc. All of this information is directly related to the execution model of the language and, because of that, costly to obtain. We think that this information would be useful to profile a parallel lazy program but it requires saving a very detailed profile of the execution of the program. Current tools cannot do this without affecting the program execution in a way that renders the results meaningless to its original purpose; that is, the information obtained is valid, but it would only apply to a program which behaviour has significantly changed from the one we were trying to debug.

Our solution to this problem is to apply the well-known technique of execution replay for designing a profiling framework with minimal interference on top of which to build improved tools for understanding parallel performance.

Execution replay (ER) [21, 22] is a debugging technique designed to use with concurrent programs. Because of its nondeterministic nature, each time a concurrent program is run would result in a different interleaving of runtime events, and possibly different code paths being executed. ER allows the programmer to record the execution trace of a program and then use that trace to replay it step by step. The trace of the program encapsulates part of the state of the system as it changes throughout the execution, so that the replay of the program can simulate the original execution as thoroughly as possible. 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.

The novelty of our approach relies on using the basic design of ER, but changing some of its requirements to make it suitable for performance profiling. In our design, the repeated execution of a program is simulated in a way that allows us to *i*) reproduce the conditions that led to the original 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 that we do during replay in order to obtain the needed profiling information without changing the runtime behaviour of the program.

Having a mechanism to obtain this information, we enable the development of more complex profiling tools than the ones available up to now. In particular, it should be possible to extend costly sequential tracing debuggers, provide heap profiles assigning costs per-thread or per-core, or create a library of new language primitives that allow to track the lifecycle of source expressions as they are evaluated.

1.3 CONTRIBUTIONS

In the development of this thesis, we have made a number of contributions:

1. We have analysed the runtime execution of a parallel purely functional language and identified the source of the nondeterminism responsible for different behaviour in different executions in the form of runtime events (listed in section 4.2.2 and table 2).

2. We have provided an implementation of ER tailored to the needs of a profiling framework, instead of as a debugging facility (described in section 4.2), that allows to overcome some limitations present in other profiling tools.

3. We have made the first implementation of ER in a purely functional language, taking advantage of the language properties, and analysing its pros and cons (presented in section 4.2).

4. We have successfully used this new profiling mechanism to profile the performance of some Parallel Haskell programs, obtaining better data about its runtime behaviour and improving its parallel performance with this knowledge (presented in chapter 5).

2

PARALLEL PROGRAMMING AND PROFILING

In this chapter we analyse the challenges that parallel programming introduces when profiling. We start by motivating the need for specialised profiling tools that target parallel execution in section 2.1 and then, in section 2.2, introduce the topic of profiling with a few definitions. Later, we review some of the more popular profiling tools developed for mainstream imperative programming languages in section 2.3. Lastly, we present an overview of a debugging technique called execution replay in section 2.4.

2.1 PARALLEL PROGRAMMING

We have already introduced the topic of parallel programming in the first chapter. We want now to make the distinction between concurrency and parallelism clear. The purpose of parallelism is to make a program run faster by using more than one processing unit. This means that a parallel program would be designed to use multiple tasks to do its work either implicitly or with more explicit approaches as was mentioned in section 1.1.1. Concurrency, on the other hand, is an abstraction mechanism to design programs that need independent threads of control (e.g. GUIs, agent systems) [10, pp. 7–8]. Concurrent programs can benefit from running on multiprocessors, but running in a single processing unit would be fine. In contrast, it would not make sense to run a parallel program in a single CPU.

Although the analysis and debugging of concurrent applications is a related subject, our thesis is focused on parallelism. In particular, semi-implicit parallelism on non-strict functional languages.

The challenges that parallel programming presents are quite different from those of sequential execution. While profiling a sequential program

is usually related to finding hot spots in the code and optimising the algorithms used or adapting them to take into account memory caches and the like, parallel programming introduces many more problems: identification of parallel tasks, lock contention, communication delays, etc. We will come back to this topic in chapter 4.

2.2 PROFILING PARALLEL PROGRAMS

Profiling is a kind of dynamic program analysis used to obtain a number of metrics about the execution of a program, for example memory usage or function execution times, with the intention of improving its performance. Based on how data is collected, we can classify profiling techniques as event-based, sampling-based, instrumentation or simulation.

Event-based profilers gather runtime metrics when certain events in the execution occur. For example, function calls can trigger the execution of the profiler to build a call graph of the program execution. This type of profilers can have a high overhead because they track every event related to the data that is being collected. *Sampling-based profiling* reduces the performance impact by inspecting the program at regular intervals using operating system interrupts. The final results are extrapolated using a statistical approximation. While this technique may seem inferior because it lacks accuracy, its results may be closer to the actual performance of the program because of its lower impact. *Instrumentation* is a technique that adds the profiler code to the program itself, either in source or binary form. Besides possible performance changes produced by modifying the program being measured, the new instructions may introduce bugs in the program. Lastly, a *simulator* is usually a hypervisor used to run the program unmodified, where the usage of a virtualised environment makes it easier to inspect the runtime behaviour of the program.

The word profiling is generically used to describe any technique or tool that aids in the purpose of program optimisation, but it is helpful to make the distinction between a profiler and a tracer, that is usually applied in practice. The output of a profiler, a profile, usually shows a

summary of the performance characteristics of the execution. Examples of those can be function execution times or number of cache misses. Profilers usually present those performance metrics together with semantic entities of the program.

Tracing, on the other hand, records chronological information about the execution of the program, triggered when a particular event or action occurs. It can be seen as a form of instrumentation-based profiling. A trace or event log usually consists of a list of records, each having its own timestamp, an identifier that specifies the type of event and additional event-specific information [23]. Event logging is specially used in the context of parallel programming because parallel performance problems are usually related to the interactions between the different tasks. At the same time, it is the most invasive form of instrumentation because of the need for saving all of the gathered information to disk and the large volume of data being collected.

The problems faced by imperative programming languages regarding software tracing are very different from the ones that affect functional programming languages, the major challenge being scalability limitations to collect and store performance data. In any case, in the rest of the chapter, we will present some examples of imperative profilers to serve as examples to give an insight of some of its capabilities. Functional profilers will be explored in detail in chapter 3.

2.3 PROFILING TOOLS

There are multiple profiling tools available for imperative programming languages. We will limit our review to a few of them used in two popular parallel programming models: MPI [4] as a representative of a message-passing distributed memory model and OpenMP [3], used in shared-memory architectures.

2.3.1 *mpiP*

mpiP [24] is a statistical profiler for MPI applications. It has considerably less overhead than other profiling tools because of it generates statistical data rather than tracking all MPI calls. Given its focus on distributed applications, the output report only summarizes information related to MPI calls and messages (e.g. execution time, message sizes, etc.).

2.3.2 *Vampir / Score-P*

Vampir [25] is a visualisation tools designed to work with different trace formats. In particular, it was originally developed to consume traces of MPI messages [26], but nowadays supports both OpenMP and MPI applications. Besides offering a general timeline view that shows processes over time, the functions they were running, and the communication between them; it also provides a set of predefined statistical charts that show accumulated measurements, such as accumulated time across functions and processes, or the communication matrix, that is, statistics on the messages sent between processes [27].

Score-P [28, 29] is the recommended tool to collect traces consumable by Vampir. It consists of a code instrumentation framework, several runtime measurement libraries and some helper tools. Its multiple storage formats allow integration with other visualisation tools such as TAU [30] or Scalasca [31, 32].

2.3.3 *ompP*

ompP [33] is a instrumentation-based profiling tool for OpenMP. It works using a source-to-source transformation that inserts calls to a monitoring library around each OpenMP construct. It work similarly to mpiP, reporting information related to OpenMP constructs, although the user can also mark arbitrary source code regions to add them to the resulting profile. Besides, general performance execution information, it can also perform overhead analysis on different areas (synchronisation, load im-

balance, thread management and limited parallelism) and detect other common problems such as contention for locks or critical sections [34].

2.3.4 *TAU*

TAU (Tuning and Analysis Utilities) [30] is designed as a framework that integrates profiling components through well-defined interfaces and data formats. Among others, it integrates with OpenMP using a preprocessor. It provides different profile variants ranging from aggregated statistics flat profiles to more complex callpath and calldepth profiles and phase profiles, that allow user-defined program state definition to classify profiling information [35]. Besides aggregation of performance metrics, it supports event tracing and includes its own visualisation tool, ParaProf [36].

Although there are many more profiling tools and libraries available in the domain of imperative programming languages, we have presented a few representative cases for two specific programming models. In comparison with functional programming profilers, the ones analysed here are mature tools that focus on pragmatic issues such as runtime overhead and providing multiple visualisation capabilities. As will be seen in the next chapter, profiling purely functional languages is an ongoing research topic that focus on the more fundamental issue of how to obtain information that is useful for profiling.

2.4 OVERVIEW OF EXECUTION REPLAY

In the same way that concurrent execution makes the process of profiling a program harder, chasing bugs in multithreaded applications is considerable more difficult than doing it in sequential applications. The main problem with concurrency bugs is that they are very difficult to reproduce. Given the nondeterministic nature of concurrency, different executions may exercise different code paths of the program each time it is run, making the classic edit-compile-test cycle not very useful. Incor-

porating additional logging statements or running the program through a step-by-step debugger to help identify the source of the problem has additional impact in how the program is executed, that is, decreases the chances of reproducing the same conditions that led to the bug under inspection. One of the techniques developed to help in that task is execution replay.

Execution replay (ER) [21, 22] is a technique designed to record the execution of concurrent applications and to allow deterministic replay of those executions as many times as desirable. Its main use case is that of debugging and it can be used in many ways [37]:

REPRODUCE NONDETERMINISTIC BUGS: Bugs that occur seldom can be reproduced if they were previously logged by the replay system.

REPRODUCE REMOTE SITE FAILURES: Given an efficient recorder, production software can be run in record mode so that users can report bugs together with the trace needed to reproduce them remotely.

TIME TRAVEL DEBUGGING: Used in conjunction with standard debuggers, execution replay can offer additional capabilities such as step backward, reverse breakpoints and reverse watchpoints. This new functionality allow to use an step-by-step debugger backwards [38].

OFFLINE DYNAMIC ANALYSIS: Many dynamic analysis techniques require costly simulations to obtain its results [39]. Performing these offline analysis in a replayed execution allow to obtain reliable data about the program execution.

As said earlier, ER systems have two phases: record and replay. The recorder is in charge of logging two types of events: *nondeterministic input* from system calls, interrupts, instructions that read the processor state, I/O, etc. and *nondeterministic memory access* in multithreaded applications in the form of shared-memory data reads and writes [40].

The recording phase is a key part for a ER system to be effective. It needs to be lightweight so that the program is affected minimally and, at the same time, complete in that all the information needed for a successful replay needs to be recorded.

The different ER systems can be classified according to their recording technique [40]:

1. *Synchronisation race approaches*, where the recording overhead is limited by only recoding scheduling decisions and synchronisation operations. These techniques provide very low overhead at the cost of not being able to correctly replay data races. Provided that the program is data race-free, replay would proceed normally, in other cases, the replay is only guaranteed to be correct until the first data race occurs. Multithreaded programs executed in uniprocessors would also work.

2. *Data race approaches*, which log all shared data accesses. Doing so has a very high overhead so most approaches use hardware support. In this description we will focus on software-only tools.

2.4.1 *Synchronisation race approaches*

liblog

liblog [41] is a library-based approach that implements execution replay for usage in distributed environments. It works by intercepting calls to the standard C library and logging their results, and also logging the contents of all messages interchanged between processes. Thread interleaving is logged by imposing a user-level cooperative scheduler on top of the OS scheduler, and only allowing context switches at `libc` call points. liblog's major limitation is that its implementation does not allow for applications with nondeterministic sources outside of `libc` API calls to be replayed.

Instant Replay

In Instant Replay [42], all interactions between processes are modelled as operations on shared objects. Modifications to those objects are represented as a totally ordered sequence of versions. To record process

interactions, a partial order of the accesses to each object is logged. Semaphores, message passing or event shared memory can be used as the objects for process interaction. The only limitation Instant Replay imposes is that the set of operations on each shared object must have a valid serialisation, requiring the use of concurrency protocols. Because only these operations are reproduced, it does not support data races. Performance overhead varies greatly depending on the granularity of the synchronisation operations.

2.4.2 *Data race approaches*

SMP-ReVirt

SMP-ReVirt [43] is a virtual-machine based replayer. It tries to minimise logging overhead by using hardware page protection to detect shared-memory accesses. The drawback of setting the granularity to the page level is that fine-grained concurrency can have a very high overhead. The authors report a runtime overhead of up to 10x on just two cores.

ODR and PRES

ODR [44] and PRES [45] are probabilistic execution replayers. Both systems record partial traces of the execution. In particular, they omit any logging of shared-memory accesses. ODR is based on output determinism: it considers a replay successful if the output of the replayed execution matches the one of the original execution. To replay the full trace, it uses a depth-first search on the space of possible outputs until a match is found. PRES works in a similar fashion, and, additionally, it has different recording modes that allow less overhead to use in scenarios where data races are not present or on single-threaded applications.

In summary, available ER implementations offer a way of fixing bugs in multithreaded applications by reliably reproducing the code paths that led to those bugs. Different work report around 10-100x overhead for state-of-the-art software-only recording of multithreaded applications [45,

46]. The main drawback of those solutions is that, while they may provide lightweight trace recording, this is done at the cost of either imposing task granularity limitations and/or failing to replay data races. In some of those cases, the replay of the program can take and indeterminate amount of time while exploring the space of possible replays. A consequence of this drawback is that applying this technique to executions with fine-grained parallelism is not supported.

3 PARALLEL HASKELL AND PROFILING TOOLS

In this chapter we give a thorough description of the GHC compiler (section 3.1), giving enough details to serve as a reference for the description of our profiling system in chapter 4. Additionally, we review previous approaches to both sequential and parallel debugging and profiling tools developed for Haskell in section 3.2.

3.1 THE GLASGOW HASKELL COMPILER

The Glasgow Haskell Compiler (GHC) [47] is the most popular compiler and runtime system implementation of Haskell. It is a free software project available at `http://haskell.org/ghc`. After its initial development by Kevin Hammond in 1989 in another lazy functional language (LML [48], its first stable release was in 1992 as a completely rebuilt standard Haskell implementation written in Haskell (except a few parts) [49]. From a plain Haskell implementation with a few extensions, GHC has incorporated in its more than 20 years of existence an interactive environment (GHCi), concurrency [50, 51], transactional memory [52], Template Haskell [53], plenty of type system extensions [54–59] and even a package management system [60].

For the purpose of understanding this thesis, in this section we provide and overview of GHC's compiler and a more in-depth look at its runtime system, where most of the implementation of our work is found, as discussed in detail in chapter 4.

3.1.1 *The GHC Compiler*

The GHC compiler has a long pipeline in which the input Haskell source code is subsequently transformed into three different intermediate lan-

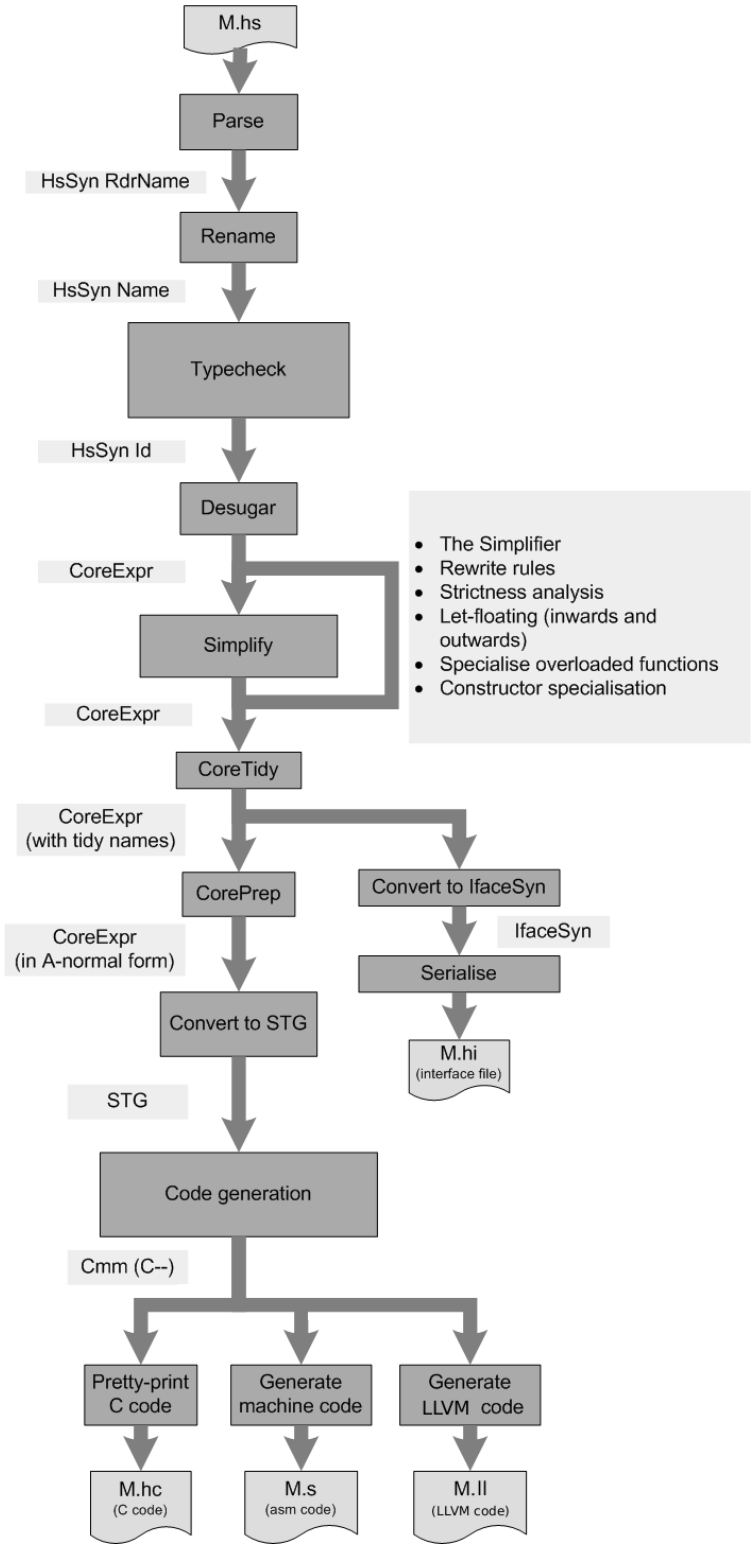


Figure 1: The GHC compiler pipeline

guages before generating the final machine code (see figure 1[1]). Before any transformation is applied, the type checker works directly on the Haskell code before the desugarer intervenes. This allows for quite detailed and informative type error messages to be presented. After type checking, the language is translated to Core [62], an explicitly typed intermediate language based on an extension of System F [63]. This is a much simplified language that, at the same time, allows to represent all the type-level features available in Haskell. After this point, most of the optimisations happen in a series of analyses, rewrite rules [64], inlinings [65], etc. The Core language is then translated to the Spineless Tagless G-machine (STG) language [66]. The STG language is a special form of Core more appropriate for direct translation to a low-level language. Then, the last language before machine code generation is Cmm. Cmm[2] is an implementation of the C-- language [67] embedded in GHC. It is a low-level imperative language inspired by C but meant to be used for compilation of high-level languages: it provides the ability to return multiple values in registers, it implements tail calls and it has an explicit stack, among other things. A different optimisation process is then applied to Cmm, mainly related to control flow [68]. Finally, Cmm is translated directly to machine code. There are other backends available that use LLVM's bytecode and compiler [69, 70] or simply use a C compiler after pretty-printing Cmm as C.

In relation to the compiler, our work involved a number of changes to code generation which will be described in chapter 4.

3.1.2 *Runtime system*

The runtime system (RTS) is the part of the compiler that implements language facilities not related to the programming language semantics, and used to interact with the operating system: the garbage collector, concurrency, exceptions, the interface with foreign code, etc. The GHC

1 Image taken from Marlow and Peyton Jones [61].

2 https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CmmType.

runtime system is a huge code base that amounts to more than a third of the size of the compiler.

Following Marlow and Peyton Jones [61], the RTS main components are 1) memory management, 2) thread management and scheduling, 3) primitive operations and 4) a bytecode interpreter. In this thesis, we focus our attention on 1 and 2. In addition, a specific section is devoted to the description of the event logging mechanism, which implements software tracing at the RTS level.

Because of the diverse functionality provided on the compiler and the RTS, much of it is compiled under what is called *ways*[3]. For example, threading support is only enabled when using the `-threaded` compiler flag, that links the program with a version of the RTS library that supports concurrency. This allows to impose no performance penalties to programs that are meant to be run single-threadedly. As well be described, the implementation of our work is also implemented using a new way in the RTS and compiler.

Memory management

GHC has a block-based parallel generational-copying garbage collector [71]. The implementation details of the garbage collector are not relevant for understanding this dissertation so we will only give a short description and concentrate on the runtime objects layout used in the execution of a program.

The garbage collector implements a generational scheme [72] with two generations. The young generation has a per-thread private nursery where new objects are allocated. To support growing and shrinking of memory areas, the garbage collector uses a block-structured heap organising memory as a linked list of memory blocks.

Any pointer living in the heap must belong to one block and a simple *O(1)* operation is used to calculate the beginning of the block where the

3 https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/CompilerWays.

block descriptor is stored. The block descriptor is a structure with all the information needed to operate with the block[4]:

```
1 typedef struct bdescr_ {
2
3     StgPtr start;          // [READ ONLY] start addr of memory
4
5     StgPtr free;           // first free byte of memory.
6     ...
7     struct bdescr_ *link;  // used for chaining blocks together
8     ...
9 } bdescr;
```

Each block has a pointer to the starting address of its memory (`start`) and another pointer to the first free byte (`free`). Because all blocks have a fixed size, there is no need to save the a pointer to the end of the block. Additionally, blocks are linked together (`link`).

Before running any Haskell code, a thread will load its heap pointer (Hp) from the `free` pointer from the current nursery block and the heap pointer limit (HpLim) from a pointer to the end of that same block. To allocate objects, the thread only has to bump the Hp with the object size after doing the heap check (checking if there is enough space). In the compiled code, every function starts with a stack (line 4) and heap check (lines 6–7) as shown in the following code listing 2 taken from the Cmm output of the parallel Fibonacci implementation from listing 1:

Listing 2: Function prelude of a `parfib` expression

```
1 n1_s15A_entry() {
2     c16r:
3         ...
4         if ((Sp + 8) - 24 < SpLim) goto c16s; else goto c16t;
5     c16t:
6         Hp = Hp + 24;
7         if (Hp > HpLim) goto c16v; else goto c16u;
8     c16v:
9         HpAlloc = 24;
```

4 https://ghc.haskell.org/trac/ghc/browser/ghc/includes/rts/storage/Block.h?rev=ghc-7.8.4-release#L88.

```
10          goto c16s;
11      c16s:
12          ...
13          call (stg_gc_enter_1)(R1) ...
14      c16u:
15          ...
16          call parfib_info(R2) ...
17  }
```

The `stg_gc_..` family of functions end up calling `stg_gc_noregs`[5], shown in listing 3 which is the function performs the real stack and heap check. This function is defined in the RTS and needs to discern whether the thread yielded (line 8 and 17) or whether a stack (line 28) or heap overflow (lines 21 and 26) happened:

Listing 3: Heap/stack check after the current nursery block is full

```
1  stg_gc_noregs
2  {
3      W_ ret;
4
5     if (Hp > HpLim) {
6          Hp = Hp - HpAlloc/*in bytes*/;
7          if (HpLim == 0) {
8                  ret = ThreadYielding;
9                  goto sched;
10         }
11         if (HpAlloc <= BLOCK_SIZE
12             && bdescr_link(CurrentNursery) != NULL) {
13             HpAlloc = 0;
14             CLOSE_NURSERY();
15             CurrentNursery = bdescr_link(CurrentNursery);
16             OPEN_NURSERY();
17             ...
18             jump %ENTRY_CODE(Sp(0)) [];
19             ...
20         } else {
```

5 https://ghc.haskell.org/trac/ghc/browser/ghc/rts/HeapStackCheck.cmm?rev=ghc-7.8.4-release#L85.

```
21              ret = HeapOverflow;
22              goto sched;
23          }
24      } else {
25          if (CHECK_GC()) {
26              ret = HeapOverflow;
27          } else {
28              ret = StackOverflow;
29          }
30      }
31    sched:
32      PRE_RETURN(ret,ThreadRunGHC);
33      jump stg_returnToSched [R1];
34  }
```

Additionally, it can also be the case that the current nursery block is full but there are still blocks to be used. In that case, after checking the size of the allocation and the block availability (lines 11–12), the current `free` pointer is stored in the block using the Hp value (line 14), a new block is set as the current nursery block (line 15) and the new values for Hp and HpLim are loaded from that block (line 16). Then, the thread continues running (line 18).

The memory layout of objects in the heap is shown in figure 2[6].

Although the word *closure* should only be used to refer to an object representing a function and its free variables, it is used in the GHC codebase to represent the common structure of any heap object[7][8] and we use it here in the same manner:

```
typedef struct StgClosure_ {
    StgHeader   header;
    struct StgClosure_ *payload[];
} *StgClosurePtr; // StgClosure defined in rts/Types.h
```

6 https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects?version=33/#HeapObjects.

7 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/Closures.h?rev=ghc-7.8.4-release#L80

8 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/Closures.h?rev=ghc-7.8.4-release#L53

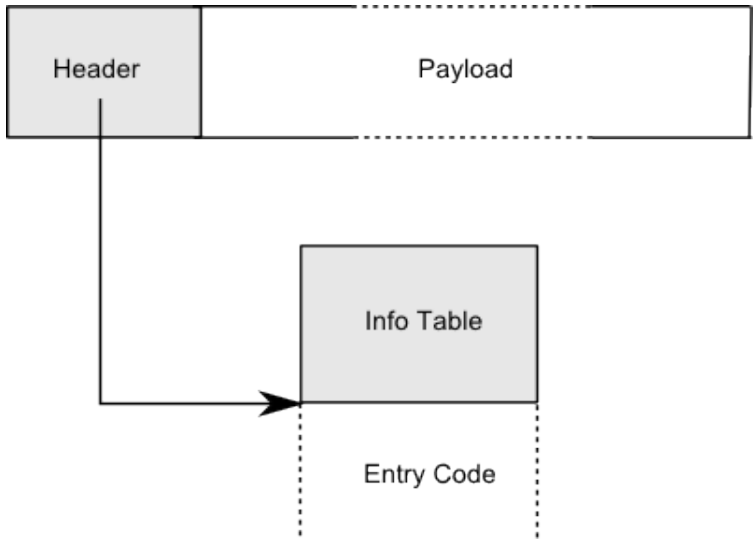


Figure 2: Layout of objects in the heap

```
typedef struct {
    const StgInfoTable* info;
    ...
} StgHeader;
```

Every closure has a header and a payload. The header is used to identify the object's type and entry code. The object's type is defined in its info table and, together with the entry code, resides in static memory and is generated at compilation time. The info tables are usually compiled next to the entry code to save one indirection when *entering* (evaluating) a closure. This works by saving in `info` the info pointer, that actually points to the closure entry code. To get access to the info table, the function `get_itbl(StgClosure *)`[9] is used. As examples, the code for a data constructor returns immediately to the topmost stack frame because it is already in weak head normal form (WHNF) and its payload contains the fields of the data value. In the case of a function closure, its

9 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/ClosureMacros.h?rev=ghc-7.8.4-release#L85.

entry code would be the compiled function code and its payload would store the funtion's free variables. The function arguments would either be passed in registers or in the stack.

The size and layout (which fields are pointers and which are values) of the payload depends on the closure type, but using an unsized array in its definition allows to cast any other object to a `StgClosure` and use the same structure fields for indexing.

We are especially interested in one kind of closure: *thunks*. Thunks are used to implement lazy evaluation by being allocated to store the result of an expression before actually evaluating it. They work in the same way as a function closure with two differences: thunks have no arguments and they can be updated. When the result of the expression is needed, the thunk is entered and the resulting value (in WHNF) will be eventually written to the thunk header. Entering an evaluated thunk will just return the result. To avoid locking when entering and updating thunks in a parallel program, they have an additional word reserved in its header to store the result[10][11]:

Listing 4: Thunk layout

```
typedef struct {
    StgThunkHeader  header;
    struct StgClosure_ *payload[];
} StgThunk;
```

```
typedef struct {
    const StgInfoTable* info;
    ...
    StgSMPThunkHeader     smp;
} StgThunkHeader;
```

By using a common closure representation and the carefully selected thunk encoding, indirections and blackholes [73] (which share the same

10 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/Closures.h?rev=ghc-7.8.4-release#L85

11 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/Closures.h?rev=ghc-7.8.4-release#L60

structure) can be directly written over a thunk to mark them as updated or under evaluation. The indirection structure[12] is shown here:

```
typedef struct {
    StgHeader   header;
    StgClosure *indirectee;
} StgInd;
```

When a thread finishes the evaluation of a thunk, the result is written next to the thunk header. Then, the thunk info pointer is overwritten with a blackhole. Later, when another thread enters the blackhole, its entry code will inspect the result. Because a blackhole looks exactly as an indirection (`StgInd`) and the indirection points to the first word after the header, evaluating it would return the thunk's result. GHC implements an optimisation called *dynamic pointer tagging* [74] that uses the last bits from closure pointers (that are always zero because they are word-aligned) to encode the value of data constructors and avoid branch mispredictions in `case` expressions. The pointer tag is also used to decide if a blackhole has been updated (points to a value in WHNF) or is still being evaluated.

Understanding how memory layout and management is implemented in the RTS is relevant as to how and why of some design decisions in our profiling system were taken in chapter 4.

Thread management and scheduling

The GHC RTS has very flexible concurrency support that is the result of many years of incremental development [50, 75, 76]. Briefly, the RTS provides a lightweight thread model where multiple user-space Haskell threads are mapped into one single OS thread which runs concurrently with others. Haskell threads can migrate between different OS threads, and tasks (understood as parallelisable units of work) are pushed to idle OS threads to increase parallelism. The concurrency subsystem is built over three different abstractions: capabilities, tasks and threads. Parallel-

12 https://ghc.haskell.org/trac/ghc/browser/includes/rts/storage/Closures.h?rev=ghc-7.8.4-release#L118.

ism, on the other hand, is handled with the use of sparks. The organisation of these concepts and their relations is represented in figure 3.

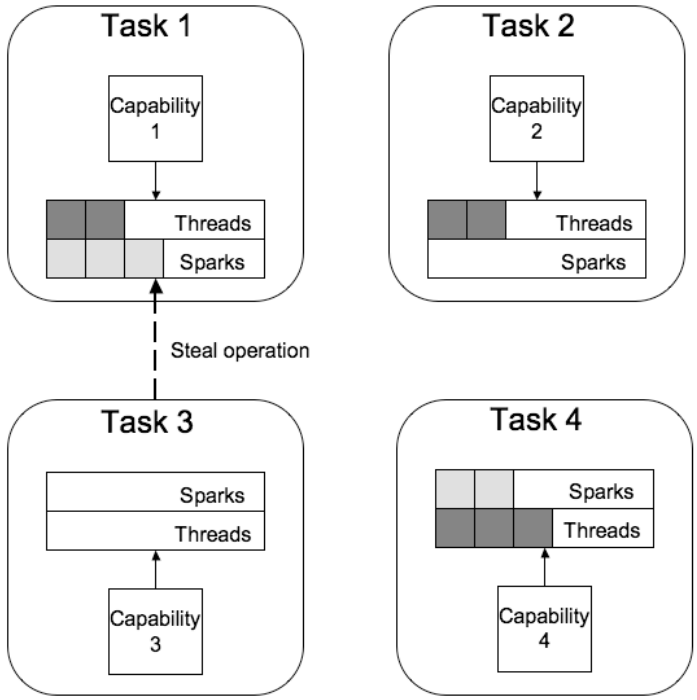


Figure 3: Overview of the GHC runtime System

A *capability* is a virtual core in which Haskell code is run. Each capability has all the state that an OS thread needs to run a Haskell thread: its private allocation area, the underlying virtual machine register table, the thread run queue, etc. 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. Those are represented by *tasks*: each task corresponds to an OS thread and will try 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. *Foreign calls* (calls to C functions, for example) that can potentially block a task (and, because of that,

prevent a capability from being available to other threads) will make the task relinquish its capability and wait in the capability's `returning_tasks` queue when finished.

There are two type of threads: *bound* and *unbound* threads. A thread becomes bound to a task when that OS thread makes an *in-call* from C to Haskell (for example, the first thread which runs the function `main` , called from the RTS). From there on, that thread will only be run by the same task. Unbound threads (created with `forkIO :: IO ()`) have no such restriction and can be run by any task. The idea behind this concept is to support the usage of C libraries that use thread-local data and therefore require calls involving that data to be executed by the same OS thread (e.g. OpenGL).

Similarly, tasks can be either *bound* or *worker* tasks. A bound task is the task that a bound thread got attached to, and will only be able to run that Haskell thread. Worker tasks are automatically created by the RTS for idle capabilities and can run any unbound thread.

In a typical concurrent Haskell program, threads created with `forkIO` would be migrated to an idle capability and be run by the worker task owning the capability. Parallelism, on the other hand, is introduced with *sparks* [76]. A spark is an abstraction used to introduce speculative parallelism by marking thunks as possible future work that can be evaluated in parallel with the main computation. Worker tasks with no thread to run will create a *spark thread*. This is a Haskell thread that looks for available sparks to run at its own capability or steals them from other capabilities. When it cannot find any spark, or there are other threads to run, the thread finishes. This mechanism is more efficient than creating a new thread for every spark that is executed [77].

A task currently running in a capability releases it under one of the following circumstances [51]:

- A foreign call is made.
- Another task is waiting in the capability after returning from a foreign call or waiting to do an in-call.

- The next thread to run is not compatible with the current task: bound thread and worker task or unbound thread and bound task. In the first case, the capability is directly passed to the corresponding task.
- The Haskell thread bound to the current task terminates. In this case, the thread would return from the scheduler to the foreign call that called it.

In any of these cases, the task releasing its capability will be enqueued in that capability and blocked until another task hands it over to it.

THREAD SCHEDULING We mention that each capability has a run queue and it is the task owning the capability the one in charge of scheduling available threads. This is performed by the *scheduler loop*, or just *scheduler*, the function that any new task runs just after being created. A very simplified version is presented here[13]:

Listing 5: Pseudocode for the thread scheduler

```
1  schedule(cap)
2  {
3    for (;;) {
4      scheduleFindWork(cap);
5      schedulePushWork(cap);
6      yieldCapability(cap);
7      tso = popRunQueue(cap);
8      result = StgRun(tso);
9      case result of
10       out of heap -> re-enqueue tso; call GC;
11       out of stack -> enlarge tso; re-enqueue tso;
12       time expired -> put tso on end of queue;
13       finished ->
14         if (tso is a bound thread)
15           return;
16         else
```

13 Based on the pseudocode from https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler.

```
17              continue;
18      }
19  }
```

The scheduler uses a simple round robin algorithm to context-switch between threads. First, it checks for new work by creating a spark thread if there are no threads to run but at least a spark available in any capability (line 4). Then it will migrate threads to other idle capabilities if it has more than one thread to run (line 5). If there is a condition to release the capability, as listed earlier, it will do so (line 6). This would give control of the capability to another task that would continue running its scheduler from the same point. Then, the first thread in the queue is taken and run (lines 7–8). When the thread gives control back to the scheduler, it is enqueued again in the run queue. Depending on why the thread stopped, it can be put on top of the queue so that it will run again (heap and stack overflow, lines 10–11) or on the back so that it will have to wait for all of the other threads in the run queue to run (time slice ended, line 12).

A Haskell thread cannot be preempted at an arbitrary moment, it needs to be so at specific points where it is safe for the thread to halt its execution. Fortunately, such safe points already exist: heap checks, where the thread is ready to stop running and start a garbage collection. The context-switching procedure takes advantage of the fact that functional programs have a big rate of allocation, so that heap checks occur frequently enough. To force a thread to yield, the RTS sets up a timer that triggers every 4ms by default and resets the HpLim register. This would make the heap check fail and, as seen earlier, the code for `stg_gc_noregs` will check HpLim and yield the thread (lines 7–10 from listing 3).

PARALLELISM As described in section 1.1.2, Parallel Haskell [14] provides two basic primitives for parallel programming: `par` and `pseq`. When a thread evaluates the expression `p `par` q`, a spark for the thunk created for `p` is saved in the *spark pool* of the capability on which the thread is running, and execution will continue with the evaluation of `q`. In other words, a spark is just a pointer to the thunk. Eventually, given enough processing power and an idle capability available, the spark will be

picked up by that capability's spark thread. In other cases, the spark may be garbage collected if the thunk it points to is not needed, or discarded if it points to an already updated thunk (fizzled, in GHC terminology). This last process of cleaning the spark pool is done as the latest phase of GC, so that they are updated before threads resume their execution.

Spark pools are implemented as work-stealing dequeues [78] to provide efficient concurrent access by threads running in different capabilities. They have a size limit so that additional sparks will simply be ignored.

The sparking mechanism is based on lazy evaluation and purity. A thunk can be eagerly evaluated in parallel because it represents a pure expression with no side effects. This makes sparks cheap to implement and therefore to provide a deterministic parallel programming model, in opposition to threads. The results from evaluating a thunk in parallel are naturally "communicated" back to other threads with the thunk update procedure. When a thunk has been evaluated, its result is written to the thunk header and, after that, the thunk is overwritten with an indirection. This step consists in updating the closure info pointer, so that the closure type is changed, and now it will be interpreted as a pointer to the result, as already mentioned in section 3.1.2. Because thunks are updated only after its evaluation is finished, it would be easy for two threads to enter the same shared thunk and duplicate its evaluation. Being the case that thunks represent pure expressions, evaluating the same expression many times would not affect the program result, but it would result in a loss of sharing and waste of resources. Blackholing is used to help mitigate this problem.

Blackholing was originally introduced as a mechanism to prevent a space leak while updating a thunk [73]. A sequential implementation of lazy evaluation would overwrite the thunk header with a blackhole closure when the thunk was under evaluation. This would allow for any object originally pointed by the thunk to be garbage collected. A nice side effect is that infinite loops would be detected by entering an already blackholed thunk. In GHC's parallel RTS, blackholes are reused as a synchronisation mechanism [75]. A thread that enters a blackhole

does not produce an infinite loop failure, it is instead put in a queue attached to the blackhole until the thunk is updated.

The problem in a parallel implementation, is that ensuring thread-safe overwriting of a closure header requires the use of expensive compare-and-swap atomic instructions. The GHC implementation uses a technique called *lazy blackholing* that delays the use of these primitives to the point where threads are paused (`threadPaused()`[14]), just before returning control to the scheduler as described in section 3.1.2. By doing this, small thunks that are updated before pausing the thread do not pay the synchronisation penalty. Long running thunks will be blackholed only after the thread has finished its time slice or has met another interruption condition.

The blackholing procedure consists in overwriting the thunk being evaluated with a blackhole closure, and in writing, as indirectee, a pointer to the thread doing the evaluation. These operations are done in reverse order and using a write barrier to ensure that a thread finds a valid pointer when reading a blackhole. When the evaluation of the thunk is finished, the thread updates the blackhole to point to the thunk's result. In both cases the blackhole works as an indirection: a second thread entering the blackhole uses pointer tagging to decide if the closure pointed by the blackhole is evaluated. In that case, it directly returns the indirectee as its result. Otherwise, there is the possibility of the blackhole pointing to another thunk, which the current thread will enter, or still pointing to the thread evaluating it. In this last case, the current thread will be enqueued in a blocking queue structure that will replace the thread pointer from the blackhole. Additional threads will be enqueued in the same structure, and later woken up when the thunk is updated.

It can be the case that multiple threads started evaluating the same thunk. Then, at `threadPaused()`, only one thread gains access to overwrite the closure, while the other threads suspend their computation by discarding any stack frames beyond the one corresponding to the thunk update. Later, those threads would resume their execution by entering

14 https://ghc.haskell.org/trac/ghc/browser/ghc/rts/ThreadPaused.c?rev=ghc-7.8.4-release#L190.

the thunk again and probably being blocked (unless the thunk was previously updated). Besides saving useless work, this also have the benefit of recovering the sharing of the expressions being duplicated. Something similar happens when the threads update the thunk being evaluated. If at that time a result is found already stored in the thunk, the thread discards its result and simply return the saved value. Alternatively, the thread may find that the thunk being evaluated was blackholed by another thread too. In this collision case, the result is overwritten and the thread also checks its list of blocking queues to wake up any waiting threads, in case one was attached to that thunk, and therefore overwritten.

Using *lazy blackholing*, the window of opportunity where different threads start evaluating the same thunk at the same time extends from the first time the thunk is entered until the threads is paused. GHC provides a compilation option `-feager-blackholing`, recommending its usage in parallel programs, that reduces this window to just one instruction [75]. This is done by making the code to evaluate a thunk write a *greyhole* before entering the thunk. A greyhole blocks other threads in the same way a blackhole does and the only chance for another thread to duplicate the evaluation of the thunk is if it reads the thunk header before it has been greyholed. The greyhole is upgraded to a blackhole just as before, at `threadPaused()`, resolving any duplicate computation.

MESSAGE PASSING Some of the mechanisms explained in the previous section require some kind of synchronisation between capabilities. For example, enqueuing different threads in a blackhole blocking queue requires only one of them being in charge. The same happens for waking up threads: they need to be enqueued in the run queue from its capability in a thread-safe way. To reduce the usage of more expensive synchronisation primitives as mutexes, GHC makes use of message passing between capabilities [76]. This procedure consists in the allocation of a message object that is enqueued in the target capability inbox using a lock. Then, the scheduler of each capability will check its inbox as part of the `schedule()` loop (listing 5).

Using this mechanism, the scenarios above are solved by sending messages to the proper capability to proceed with the corresponding task. In the case of multiple threads blocking on a thunk, they would message the capability owning the thread that is currently evaluating the thunk and then interrupt that thread. Later, after the thread is paused, it will check its inbox and create a blocking queue for the first block message pointing to the sending thread, and also enqueue there the second thread when processing the second message.

The implementation of concurrency and parallelism in the RTS is very complex and it constitutes the main source of nondeterminism in the execution of pure Haskell programs. As such, it will be very important to determine what behaviour needs to be tracked and, in doing so, chapter 4 will reference back to this section.

Event logging

The implementation of our profiling technique requires the use of software tracing at the RTS level. Fortunately, an event logging subsystem already exists in GHC. It was introduced by Jones Jr., Marlow and Singh [20] to be used with parallel profiling tools and consists of two parts:

- Support in GHC's RTS to emit events in a well-defined trace file format. This must be done in a lightweight manner so that there is minimum overhead associated with emitting an event.
- The `ghc-events` library that is able to parse the trace file and uses a Haskell algebraic type to make it easy to manipulate.

The objective of event logging is to provide runtime information for external tools to display, and to help in debugging parallel performance problems. The idea is that the events represent useful execution related behaviour such as a thread starting its execution or garbage collection being triggered. The events provided by event logging are classified in four categories: scheduler related, garbage collection related, sparks related

and user-defined events. By default, events from the first three categories are enabled. As an example in the first category, events are emitted just after a new task is created, after a task is deleted and after a task is migrated to another capability. Each event has a timestamp of when it was emitted, an event type identifier and a number of parameters depending on the event. For a new task, the `EVENT_TASK_CREATE` constant is used[15]. Additionally, this event contains the task id, the OS thread id and the capability number in which it is created[16].

In order to reduce the overhead that implies having to write this information to a file, events are also classified using another property: capability-local events and global events. Because events can be generated from independent OS threads at the same time, they are synchronised using costly mutex primitives so that they are appended to the event log atomically. Then, in order to improve on this scheme, the tracing infrastructure uses a number of event buffers: one for each capability and a global one.

We know that only one task can run concurrently with others if it owns a capability. In that case, no other task can access private data structures from the capability. Then, events stored in a capability's event buffer can be written without any lock and global events are the only ones that need to use locking primitives. Because the event subsystem was designed to report information about parallel behaviour, most emitted events belong to the first category, capability-local events.

3.2 PROFILING TOOLS FOR HASKELL

Because of the relations between each other with respect to the techniques developed for each purpose, in this section we take a look to several of the tools targeted at debugging and profiling lazy functional programs. Even though they all were built to work with the Haskell programming language, their approaches do not have anything specific to

15 https://ghc.haskell.org/trac/ghc/browser/ghc/includes/rts/EventLogFormat.h?rev=ghc-7.8.4-release#L161.

16 https://ghc.haskell.org/trac/ghc/browser/ghc/rts/Trace.c?rev=ghc-7.8.4-release#L578.

it, so they are of general interest for any non-strict language and even to strict languages in some cases.

3.2.1 *Sequential debugging/profiling*

Imperative languages have long supported multiple debugging and profiling tools for sequential execution. In their execution model, every function call pushes a new frame to the call stack allowing external tools to inspect this stack and construct a stack trace which faithfully represents the actual lexical call context. The problem with non-strict functional languages such as Haskell is that their execution model behaves very differently in three aspects: the use of lazy evaluation, tail call optimisation and extensive use of optimisations [79].

- *Lazy evaluation* makes it very hard to get information about the execution order out of an execution stack. Because the order of evaluation is driven by demand, essential information may be lost because it comes from expressions already evaluated or skipped. We can use a simple example to illustrate this [79]:

```
main = do
  [x] <- fmap (fmap read) getArgs
  print (head (f x))

f x = map g [ x .. x+10 ]

g :: Int -> Int
g x = 100 `div` x
```

  When the program breaks because of a division by zero, the execution stack looks like this: `main` > `print` > `g`. `f` is not there because it has been evaluated as needed, so it just calculated the first cons cell.

- *Tail call optimisation (TCO)* [80] is a technique that, instead of adding a new stack frame on top of the old one when calling a function, it replaces the current frame with that of the calling function if it is

a *tail call*. Tail calls are function calls executed as the last thing in a function body. In a functional language, this would correspond with the function call being the topmost expression of the function body. In those cases, the result of the caller function is going to be the result of the callee. This means that, after the callee returns its result, the caller will just return that value as its own. Because of this, the caller stack frame is of no use when doing the tail call and it can be discarded. This optimisation is essential for functional languages so that tail recursion (recursive functions where the recursive call is a tail call) can be implemented efficiently by translating it to loops. The problem with this optimisation is that the call stack is incomplete, missing many function calls. It is relevant to note that besides tail recursion, in a functional programming style it is very common to use many small functions and function composition repeatedly, leading to expression like the following:

```
main = print (f 1)

f x = g (x - 1)
g x = id (x * x)
```

In this example, inspecting the stack in the middle of the evaluation of `id` would result in the following call stack: `main` > `print` > `id`, where `f` and `g` would be missing because of TCO.

- Aggressive *optimisations* can change the executed code in a way that it is very difficult to relate back to the source code. This is particularly true for inlining [65], an optimisation which selects specific functions (usually small ones to avoid code bloat) and inlines (copies) the function body into its call sites in order to allow further optimisations to take place.

Given these three features that, each in its own way, prevent functional programs from being analysed and profiled using general tools, we proceed to classify previous tools according to its approach to profiling. We arbitrarily decided to group them in three categories: *profiling tools*, *tracing tools* and *other tools*. We assigned each category depending on the

dominating technique. That is, even if almost all of them can be classified as profiling tools, on one hand, the tools that are based around creating a log file with relevant information to postprocess afterwards, or maybe use it to drive a later debugging step were classified as *tracing tools*. On the other hand, all of the profiling tools use some kind of tracing to save the gathered data, but if the novelty of the approach relies in the profiling mechanism itself, they were classified as *profiling tools*. *Other tools* is used for ad hoc techniques or others that fall outside of the previous two categories.

Profiling tools

In this section we do a chronological review of the main different techniques and tools that were developed to do both space and time profiling for Haskell.

YORK HEAP PROFILER This profiler [81] was originally developed for LML [48] and later made available for the Chalmers Haskell Compiler (also known as HBC) [82]. It provided two features:

- It generated a heap profile that summarised the amount and type of allocated heap cells at regular intervals in the execution. It worked by modifying the compiler to attach tags to every cell in the heap. These tags were used to identify the producer of the cell (a function) and the construction that it represented (data type or specific constructor for algebraic types).
- Using the heap profile, it generated a graph of producers or constructions. This graph represented the amount of heap space taken by heap cells over time, grouped by its producer or construction.

At runtime, a clock was used to signal the time to do profiling. At that point, a flag was set so that, at the beginning of each function, a special code could identify the request and start analysing the heap (called *heap census*). This allowed to do the profiling at specific points were the heap was in a known state (no partially updated nodes).

This tool provided the programmer with information about how memory was being used by assigning real used memory values to functions in the program's source code. Additionally, command line parameters allowed to restrict the scope of the profiling to specific producers or constructions, so that one was used as a *selector* for the results showed in the other profile (e.g. restrict the producer profile to the most memory consuming construction). This was useful to focus on particular points of the execution.

Besides being able to optimise parts of the code responsible for most of the allocation, the plotting of memory usage over time was very useful to find *space leaks* –which are especially common when using lazy evaluation.

This tool is one of the first implementations of a heap profiling mechanism for lazy languages. It provided an integrated application –both the compiler and the post-processor– that allowed to fix many common programming errors but, at the same time, it presented some limitations:

- It did not consider time profiling at all.
- There was no mechanism to aggregate information. That means that producers had no relations with each other so that the caller of a function would not subsume the costs assigned to it. Also, that there was no way to differentiate different applications of the same function (it had to be renamed to mimic that result).
- The information attached to each cell was static. This prevented additional profiling in the form of dynamic attributes such as *creation time* that would give information that could explain whether a high demand for heap space happened due to a constant rate of cell replacement or due to *dragging* (cells surviving beyond the point they were last needed).
- It did not provide information about the access relations of the cells. It showed the distribution of the heap by its producer but not the reason why the memory was not being released: the part of the program that retained all that memory.

NHC HEAP PROFILER The nhc heap profiler [83] extended the York heap profiler functionality by measuring the lifetime and the closure retainers of heap objects in the nhc compiler [84, 85]. Instead of trying to find the presence of space leaks, it tried to find the causes of such programming errors.

Lifetime profiling allows the programmer to identify long-lived or short-lived cells. As an example, long-lived cells might represent large unevaluated closures for which eagerly forcing its evaluation might reduce space usage. The profiler allows to use lifetime data as a selector in the producer and construction profiles. Lifetime information is obtained by storing the creation time of each heap cell at the time of allocation and recording population counts at each heap census. A post-processor is then used to derive lifetime data from the log file.

Retainer profiling gathers informations about retainers. A *retainer* is a consumer. The name makes reference to the fact that, before a particular node has consumed some piece of data it depends on, it is retaining it from being garbage collected. This mode of profiling allows to generate graphs where the amount of data retained is labelled with its retainer, so that it answers the question "who is responsible for these cells being alive?", a piece of information much more interesting that just knowing which cells are kept alive. Similarly to lifetime profiling, retainer profiling requires and additional word in each heap cell to store its *retainer set* (sharing implies that there may be more than one retainer per cell). On the speed side, it requires many passes over the heap to find the exact retainer sets of each cell.

Experiments run in the original implementation [83] found execution overheads of several times the non-profiled execution.

As a conclusion, both lifetime and retainer profiling added very useful information to the original heap profiling method developed for LML. Retainer profiling can help in finding out which functions are responsible for dragging or closure accumulation. Lifetime profiling, while more limited, can reveal possible dragging or be used to narrow the kind of cells to investigate in other profiles. On the downside, these profiling methods impose significant overheads that make the approach limited to specific environments.

COST-CENTRE PROFILING Sansom and Peyton Jones [17] developed cost-centre profiling at the same time that the heap profiling technique used by the nhc profiler [81, 83]. Its main difference is two-fold:

1. Instead of using functions or other cell types as the basic unit to which assign costs, they introduce the notion of cost centre, a label which can be attached to any expression and captures the costs associated with it.

2. It covers both time and space profiling, providing the same heap profile graphs and a summary of cost centres timing costs.

A cost centre is assigned to any expression by using the following syntax: `scc cc e`, where `scc` is a new construct called set cost centre that assigns the string `cc` to the expression `e`. Newer versions of the compiler deprecated this ad hoc syntax introducing it with the more general mechanism of program annotations written in comments: `{-# SCC "cc" #-} e`. Using an informal description of its cost semantics, "the costs attributed to `cc` are the entire costs of evaluating the expression `e` as far as the enclosing context demands it, excluding free variables and inner `scc` expressions" [17, pp. 2].

Cost centres have a few benefits over restricting the programming language construct to which costs are attributed:

- Sansom [86] provides a formal cost semantics so that cost attribution is well defined, even in the presence of optimisations which involve code motion.

- In opposition to the York heap profiler, costs are aggregated so that the costs of callees are subsumed into the costs of the caller.

- Selective usage of cost centres allows to avoid assigning cost to many small functions or library functions that are heavily used but do not provide much information (e.g. `map`).

The implementation of this technique consists in calculating a statistical approximation of the usage share of each cost centre and the alloc-

ation they were responsible for. Cost centres are represented by a structure which keeps counters for each of the costs calculated. Whenever an expression with an attached cost centre is evaluated, a special register "current cost centre" is set to point to that cost centre's structure. After evaluation, the previous cost centre is restored. Then, a timer instructs the runtime system to periodically stop evaluation and increment the usage counter of the current cost centre. Additionally, every heap allocation is also assigned to its corresponding cost centre. At the end of the program, a summary of the execution is saved in a log file by checking the counters of all cost centres and calculating its share of the total as a percentage to represent time, and its real heap allocation.

Cost-centre profiling also produces heap profiles in a similar fashion to the York heap profiler. The implementation is very similar: an extra heap word is added to each cell to store the current cost centre at the point of heap allocation.

Comparing it with the York profiler, the main benefit of this approach is the aggregation of information around cost centres and its precise semantics. Later implementations of cost-centre profiling subsumed the previous heap profiling techniques by adding lifetime and retainer profiling.

The disadvantage of this method is that it can have overheads of around twice the original execution time. Besides introducing the overhead related to the profiling bookkeeping, some optimisations that move expressions across cost centre boundaries have to be disabled.

COST-CENTRE-STACK PROFILING Morgan and Jarvis [87] identified some problems with the previous two profiling mechanisms:

- Profiling takes a long time: limitations in the existing profiling implementations forced the programmer to repeat profiling using different options to focus on distinct sets of functions or cost centres. This was due to the static nature of profiling that, once costs are assigned to their producers there is no way of aggregate them in parent/child relations.

- Profiling results can be misleading: at the time of interpreting the results obtained, it is often useful to proceed in one of two ways, this is, either display the high level results and then decompose them into smaller pieces, or display a low-level view of the results and reconstruct the functional dependency chain of the program from them. In order to do this, there needs to be a inheritance mechanism in place to aggregate costs in a parent/child relation. The York heap profiler lacked any such mechanism and the cost-centre profiler had a limited implementation that subsumed costs from functions with no cost centre to its first enclosing cost centre.

Cost-centre stacks generalise the above aggregation mechanism by extending it to every cost-centre. By building lexical stacks of cost centres, the view of the profile can be repurposed by removing some uninteresting functions and inheriting their costs upwards without repeating the profiling.

This mechanism can be used to produce the exact same profiles as the previous ones, just by manipulating the way in which its data is presented. Additionally, it provides full inheritance profiles which associate cost centres with the entire costs of evaluating the attached expression, including any possible inner cost centre.

When cost centres are associated to every function, a cost-centre stack can be thought of as the stack that would result of the strict evaluation of the program.

This technique can be considered as a refinement of cost-centre profiling and, besides providing more information, presents the same performance characteristics.

Tracing techniques

In this section, we limit our discussion to two tools that developed their own abstractions to represent and trace lazy computations: Evaluation dependence trees in the first case, and redex trails in the second one.

FREJA Freja is a compiler for a subset of Haskell 98 [88] developed to showcase novel declarative debugging techniques for lazy functional

languages [89]. In his PhD thesis, Nilsson [90] gives a comprehensive description of declarative debugging and an efficient implementation of Evaluation dependence trees (EDTs) [89].

Freja's approach is based on the construction of a trace of the lazy evaluation which abstracts over the details of its evaluation strategy. This is done using EDTs. They consist of a tree of nodes that represent each reduction of the program execution, storing the function name, arguments and its result. This tree structure is constructed at runtime, and is used when debugging by navigating the evaluation nodes upward. It is built in a way that keeps the syntactic structure of the source code, so that it resembles a strict call tree, even though it is independent of the evaluation strategy.

Because an EDT stores every reduction that happened in the program's execution, some techniques to reduce the space footprint were developed. In particular, Nilsson and Sparud [91] use *piecemeal tracing*, a scheme where parts of the EDT are constructed on demand by re-executing the program being debugged.

EDTs are used in Freja in a question/answer interface which presents the user with a reduction step and its result to decide on its correctness. A negative answer would identify the error and its place in the source code, and a positive answer, would be followed by navigating to the parent redex and repeating the process.

As we described earlier, this debugger is implemented as part of a compiler of a subset of the Haskell programming language. The implementation is done by modifying the abstract machine underlying the compiler implementation so that the overheads can be kept to a minimum. Even so, the evaluation presented by the author shows that the re-execution with debugging turned on can take up to several times the original execution time [92].

HAT Hat [93] takes a very similar approach to Freja: a program transformation is done so that, besides computing the result of the program, a trace of the execution is constructed at runtime. The authors call it a redex trail and it is a directed graph of value and redex nodes. Before the program finishes, a browser is used to navigate the trail interactively.

The main difference with the previous debugger is the usage of a program transformation instead of an implementation inside the compiler. This has the benefit of a portability but it incurs in a much greater overhead: it takes an order of magnitude of execution time and its space usage can be of several orders of magnitude.

Techniques to lower the space usage were analysed by Sparud and Runciman [94]. They describe the construction of partial trails by ignoring the redex trails of trusted functions (library functions known to be correct) and trail pruning. The latter consists in limiting the length of the redex trails to a given number and prune larger trails at garbage collection time.

A comparison of Hat, Freja and HOOD, described in section 3.2.1, is available in Chitil, Runciman and Wallace [95]. In their analysis, similarities between EDTs and redex trails are remarked. This work led to the development of augmented redex trail structures [96], an extension of redex trails that added support to Hat for algorithmic debugging and observations, besides its original redex browser.

Other tools

For the sake of completion, we offer a description of other tools that do not belong in any of the previous categories, but contribute to give a complete overview of the functional debugging landscape available for Haskell.

HOOD The Haskell Object Observation Debugger (HOOD) [97] is based on the observation of intermediate data structures. Gill [97] identifies some problems with the usage of debugging functions like `trace`:

- The interleaving of the debugging output due to lazy evaluation and the strictness properties of the very same debugging function which can trigger other traces to run.
- The need to modify the code to debug in meaningful ways.
- The change in the strictness properties introduced because these functions usually print their results immediately.

As a solution to this, Gill [97] proposes to *observe* intermediate data structures. The implementation of this technique comes in the form of a portable library of combinators where its most useful one is `observe :: (Observable a)=> String -> a -> a`. This function can be used to mark any data to be observed. A typical use with very small footprint would be placing it in a pipeline: `consumer . observe "intermediate" . producer`. `observe` preserves the strictness of its inputs and saves them so that they can be rendered at the end of the program, solving all of the aforementioned problems.

Additional benefits of this library are the possibility of observing functions at their application sites, considering them as mappings from inputs to an output, including higher-order functions. The results of functions living in the IO monad can be normally observed, and other combinators allow to take snapshots of the state monad.

Although the evaluation strategy is not modified by using this debugging technique, the space behaviour of the program can change as the result of replicating observations, because the implementation of the combinator is done in a way that would lose sharing in this case.

Reinke [98] implemented a graphical tool on top of HOOD that provides a dynamic graphical visualisation, based on a simple tree layout algorithm.

HSDEBUG HsDebug [99] tries to implement a conventional imperative debugger that follows a stop, examine and continue model: that is, a debugger with the ability to stop the execution, examine the state of the program, usually including the ability to run functions that manipulate that state, and continue the execution.

As we described at the beginning of section 3.2.1, there are some properties of lazy languages that prevent a direct implementation of that model. HsDebug uses different techniques to deal with tail call elimination, lazy evaluation and optimisations that heavily alter code:

- *Transient tail frames* are implemented by doing tail call elimination when there is no stack space left and at garbage collection time

(the latter needed to prevent holding onto evaluated data longer than needed).

- *Optimistic evaluation* [100] is used to obtain lexical stacks from the execution. This evaluation strategy is a variation of non-strict evaluation where the program is run strictly but uses *abortion* to stop non-terminating computations. Those expressions, which can be long-running or failed computations, are then normally evaluated *as needed*, so that the call-by-need properties are maintained.
- Transformations that modify the program in ways that make it very difficult to map it back to the source code are disabled.

We can point, as a drawback applicable to many of the other approaches, to the reliance in modifying the program behaviour, both by disabling some optimisations and by completely changing its evaluation strategy. Unfortunately, the implementation of this tool has not been made available by its authors yet, making it difficult to evaluate its effectiveness.

3.2.2 *Parallel profiling*

By *parallel profiling* we refer to the fact of doing profiling of parallel programs, independently of whether the profiling process itself has a sequential nature or a parallel one.

In the same way as it happened for sequential programs, there is an overlap of the features and techniques that can be classified as *profiling* and/or *tracing*. Additionally, many of the systems analysed use simulation of the target program combined with the former. Because of this, we decided in this case to avoid any classification and give a chronological description of the relevant tools in this space.

One of the first techniques to analyse the performance of parallel Haskell programs was presented in Roe [101]. In that work, the author develops a simulation technique which is based on program transformation. A source-to-source transformation is used to gather metrics of

an idealised parallel execution. The transformed program, besides computing its result, performs an *abstract simulation* directly tuned by the programmer. To illustrate the approach, Quicksort is used with both synchronous and asynchronous parallelism. In the first case, the parallel execution is supposed to start after splitting the input list in two, and two parallel tasks would sort each sublist, ending with a sequential append. To calculate the average parallelism, the program is altered to count the maximum number of sequential comparisons in each parallel path. In the asynchronous case, pipelined parallel evaluation is used to split the input list and start the sorting process once the first element of each sublist has been calculated. Step counts are not enough to take into account asynchronous task creation, so the program is further transformed adding timestamps to each list element once they are evaluated. This marks the time a task can start its parallel evaluation of that value. With the information of tasks execution, parallelism profiles and task activity charts are available. A formal description of this technique is available in the author's thesis [102]. Because it is based on program transformation, there are many measurements that cannot be performed and, as in the case of expensive profiling techniques, the metrics gathered will be unreliable.

In 1993, Runciman and Wakeling [103] introduced one of the first approaches to profiling parallel functional programs by using tracing. They designed a tracing machinery to profile Parallel Haskell programs. Tasks were classified as being in one of three states: *running*, *runnable*, or *blocked*. Sparks were tagged with the function spawning the task and a global task counter to differentiate sparks from the same function. This information was then saved in a log file when the program was run, together with a reduction number of each event as a substitute for time. With this information, the programmer was able to generate graphical profiles of task transitions.

The implementation of this technique was done in the Chalmers HBC compiler, modified to be quasi-parallel, where the underlying G-machine was extended with multiple threads of control but the parallelism was limited to the abstract machine, and the program had no real concurrency.

Besides the downside of using a simulator instead of real parallelism, the usage of reduction numbers meant that the overhead of synchronisation (and/or communication) and task creation was completely omitted. Additionally, the graphics generated provided very limited information, forcing manual inspection of the trace.

A few years later, Hammond, Loidl and Partridge [19] developed a similar system with several enhancements called GranSim [104]. This simulator was based on the threaded runtime system of GHC. In this case, the simulation was not based on an idealised parallel machine, nor a particular one. The system was designed so that many hardware choices were parameters to be defined at the time of running the simulation (for example, packet latency and construction times). Sequential simulation was done by analysing the generated code and providing weights to each instruction type (arithmetic, loads, stores, etc.) depending on the simulated architecture.

In comparison with the previous approach, GranSim was able to gather much more data from the simulation. When run, each thread stores task state information, spark information, amount of allocation, amount of evaluated code, etc. Besides providing the same graphics with activity profiles, GranSim gives a great importance to information about thread granularity. For this, it produces profiles of the number of threads by execution time, amount of heap allocation, number of sparks and time spent in communication.

On the downside, a simulation can take a few orders of magnitude longer to run in comparison to the compiled sequential version. Additionally, even if it seems very useful for obtaining general information, it cannot be related back to the source code running in each thread, so that the previous graphs need to be interpreted to explain performance bugs and be able to fix them. The major drawback of this approach is that relying in a simulation makes the information based on the particular assumptions about the simulated cache and instruction costs.

The work on the GranSim simulator was followed by a lot of additional work expanding its features:

- Similarly to how cost-centre profiling was developed for Haskell, and GHC in particular, parallel cost-centre profiling [18] was introduced for parallel non-strict functional languages. In this work, Hammond, Loidl and Trinder [18] develop a profiling technique that combines both the GranSim simulator and cost-centre profiling, called *GranSim-CC*. The runtime system enables cost-centre profiling of each simulated processor by saving and restoring the current cost centre in the running thread, instead of globally. Then, whenever the current cost centre was changed, a new entry was added to the tracing log. The same visualisation tools of GranSim were used, with the addition of profiles broken down by cost centre.

- King, Hall and Trinder [105] describes the development of *GranSim-SP*. The idea behind this project is to assign the costs of evaluating parallel threads to the GpH evaluation strategies [14] used to create them. The implementation is an extension of the GranSim simulator and adds a new combinator `markStrat :: String -> Strategy a -> Strategy a` that assigns a label to a strategy. Then, the visualisation tools can select the thread's colour based on the label used by the strategy that created it. Child-to-parent relations are also tracked to increase the granularity of the labels shown in the profiles, i.e. two different strategies may use the same nested strategy but will create threads with a different label for each one.

- To overcome the downsides associated with the static nature of graphical profiles and the either excessive or insufficient amount of detail, Charles and Runciman [106] develop an interactive approach to application profiling where they design a query language that can be used to inspect GranSim logs and tune its graphical tools.

Unfortunately, the GranSim simulator was a profiling tool limited to the GUM compiler and runtime system implementation of Haskell [107]. That implementation was never merged to the upstream GHC sources and had limited use beyond research purposes.

More recently, there has been some work in making up-to-date tools that provide some of the functionality offered by GranSim but usable by the generally-available GHC compiler. Jones Jr., Marlow and Singh [20] developed a visualisation tool called ThreadScope that allows the programmer to interact with a graphical profile showing the main activity of the parallel execution in a timeline. This view is very similar to the activity profiles provided by previous tools and makes use of the event logging subsystem described in section 3.1.2.

3.3 SUMMARY

Many of the sequential profiling tools and techniques described in this chapter give enough detail of the program execution to obtain a complete understanding of its behaviour. The problem with these tools is that the amount of detail given is generally in direct proportion to the amount of overhead to run the modified program. The more informative tools, based on software tracing at the program level, need to record a trace of the whole execution and this incurs in very high overheads. Besides, they require non-trivial transformations of the source code or a completely different implementation of the compiler, which makes them suitable for finding logical bugs in the program, but not to search for performance problems when using the real compiler and RTS implementation. The same reasoning can be applied to HOOD or similar techniques that modify the execution of the profiled program in a substantial way.

The other sequential tools suffer from similar problems. In the case of the heap profilers, all of them present high overheads due to closure size increases done to store source code labels. Also, requiring to stop execution to perform a heap census from time to time makes this technique unusable in a parallel context. The techniques based on cost-centre profiling require modifications in code generation as well, and disable some of the performance optimisations that would prevent correctly tracking cost centres. These changes add to the overhead already paid by setting the current cost centre every time a differently-tagged expression is entered.

Parallel profilers tried to overcome the limitations described here in two ways: either by performing a simulation so that any overheads would not be paid by the execution being monitored, or by doing very lightweight tracing; so that only the basic information was extracted. In particular, GranSim is able to provide some of the information given by sequential heap profilers and cost-centre profilers in combination with a simulated environment. The problem with relying in simulations is that the parallel overhead usually attributed to cache and memory configurations and other limitations related to the specific environment in which program is executed will be calculated depending on the particular hardware configurations implemented in the simulator. When moving the program to a hardware platform, the parallel behaviour may change significantly. Event logging provides a very promising alternative in that the measured overheads are small enough to consider running the modified version of the program in production. Its downside is that the obtained data generates very general graphs that only show underperforming cores, but the lack of detail available in the other tools does not allow to identify the cause of performance problems beyond global thread granularity.

The creation of a real alternative for profiling parallel program should try to reconcile the amount of detail available in the sequential profilers with the limited overhead that parallel profilers present.

4

EXECUTION REPLAY-BASED PARALLEL PROFILING

In this chapter we describe our implementation of execution replay in the GHC runtime system as the foundation of a novel parallel profiling technique. We divide the chapter in two sections: we first describe the features of an ideal profiling tool for a parallel lazy functional programming language and the challenges that its design and implementation involve. Finally, we describe the implementation of our profiling framework detailing what design decisions have been made to accommodate it to a particular RTS of a lazy functional language in section 4.2.

4.1 DESIGNING A NEW TECHNIQUE FOR PARALLEL PROFILING

We described in chapter 3 the currently available profiling tools and their shortcomings. When surveying the tools designed for profiling parallel execution, we saw that all of them involve some form of tracing, either as additional instrumentation in the program or invoked in a simulated environment. One of the trade-offs usually made in their design is that the profiling code should be as lightweight as possible. This implies exposing a limited amount of events in their tracing system that does not perform any computation beyond gathering available data from the RTS and doing small calculations so that some statistics can be stored and reported when needed.

The information provided by these tools is valuable to have a broad understanding of the overall performance of the program:

- It makes it possible to know how much of the computing power is being used. For instance, the general view provided by ThreadScope shows how much time capabilities are being used by the available threads and how much time they are idle.

- It makes it possible to detect granularity problems. For instance, the thread profile of GranSim shows a lifetime view of each thread so that its number and computing load can be considered in comparison to other threads.

- It makes it possible to detect some data-dependency problems. For instance, the same thread profile discussed above can show if threads are blocked shortly after being created.

However, a property of all the questions we are able to answer with the data from these tools is that they are inquiries about generic details of the execution. We may know that threads are too fine-grained or coarse-grained but no tool points to the cause for that behaviour. We may get reports with all threads that block in the program, but we just know their thread ids and the ids of the threads they block on and not the source expressions in which they are blocked. This information is useful as we recognise, but it just reveals the effect of a performance bug, not the cause. All we can do is to use the insights given by these tools to try to guess how the program is actually behaving and applying some changes with the hope that they solve the problem and obtain better speedups.

We believe that, in order to solve the common performance problems of the parallel execution of a lazy functional programming language, a profiling tool needs to provide more detailed data of the runtime behaviour of the application that is *not limited to observable effects*. Such tool needs to be *tightly integrated with the evaluation model of the language and/or its internal object model* so that it can answer questions that involve understanding the semantics of the language. The following is a list of additional information that we believe would be very valuable in helping to debug parallel performance problems and is not provided by currently available profiling tools because of their design:

- *Source code correspondence for significant events*. We discussed in chapter 2 why pointing to source code from an arbitrary point in the execution is a very hard problem in a lazy setting. However, providing this ability in a limited number of cases can greatly increase the

understanding of a parallel profile by enhancing a thread lifetime view with the expressions being evaluated.

- *Introduce new events for language-related phenomena*. New language primitives could instruct the profiling system to track the lifetime of specific thunks and emit the complete enter/update cycle as events. Later, from a thread profile with this information, one would be able to deduce if a particular thunk is responsible for a sequential part of the execution or how its lifetime relates to the lifetime of other thunks.

- *Memory usage profiles* need to be available for parallel executions. The information provided by sequential memory profilers such as cost-centre-stack profiling analysed in section 3.2.1 should be available in a parallel setting, with additional information relating memory usage to each thread and capability.

- *Additional information about task granularity*. To improve the understanding of tasks granularity and help with their definition it would be valuable to provide information about the amount of sparks created and promoted to threads, spark sites producing the most valuable threads, amount of wasted work by duplicated evaluation or unused speculative parallelism (i.e. garbage collected sparks due to their results being unused).

Some of the items listed before are already present in a few of the sequential profiling tools analysed in chapter 2. One could argue that designing the profiling tool we desire for parallel programs is just a matter of translating the available sequential designs to a parallel setting. In the next section we will argue that this is not possible because of some characteristics of parallel execution that make it qualitatively different from sequential execution.

4.1.1 *The observer effect and profiling parallel programs*

In physics, the observer effect states that the act of observation will necessarily produce an effect on the phenomenon being observed. Translating this principle to the context of profiling parallel programs, one could state that:

> Any instrumentation applied to a parallel program in order to obtain some information about its execution will necessarily produce a change in that very same execution.

Being this the case, the results obtained by the instrumentation will only apply to the modified execution, and not to the one we were interested in to start with.

Obviously, this fact applies to sequential programs as well, but there are some particular properties about parallel execution that make the interference of the profiling process especially relevant. Except for some pathological programs, different executions of the same sequential program should have similar performance characteristics. This happens because the amount of nondeterminism found in a sequential program execution is generally low, restricted to operating system calls or usages of random data generators, and only programs with a very particular purpose exhibit some nondeterminism, which is usually needed for the problem being solved. On the contrary, the execution of parallel programs is inherently nondeterministic in a fundamental way. The nature of parallel execution implies that there must be multiple threads of control (though not necessarily explicit), with independent executions that will interact in some particular cases: access to shared data or message passing. Because of its independent execution, the interactions between threads will occur at different times each time the program is run. A result of this is that different executions of the program will result in different runtime behaviour and, following this, different performance results.

Given this property, it follows that observing the performance properties of sequential programs should give reasonable results by adjusting

the results for the overhead introduced by the observation process itself. In the case of imperative programs, timing individual functions is relatively easy by just comparing the time when the function is called with the time when the function ends. Doing that in a lazy functional language is quite harder, but it is a problem that has been solved by the use of cost-centre profiling (see section 3.2.1). As described in chapter 2, this kind of profiling uses statistical sampling to obtain function execution times, and relies in the instrumentation having an equally distributed overhead over the program. Other information could be obtained by invoking the needed routines either from the program (e.g. `printf`-like functions), or by enabling a debugging version of the runtime system linked to the program. In both cases, the profiling version of the program will have some amount of interference in the execution, but, because of its deterministic nature, it will be mostly equivalent to interleaving the execution of the profiling routines into the execution of the pristine version of the program[1]. As a result of this, the gross effects of the profiler can be calculated and subtracted when reporting its results.

Now, if we consider programs written in a parallel language, there is an immediate problem that results from this fact: the overhead applied to any part of the program is no longer isolated from the rest of the program. Because there are other threads running at the same time, those threads may change its behaviour in many ways. For example, if we produce a slowdown in a thread evaluating a shared thunk, we will unequally affect other threads that may be waiting for the thunk's result.

As a summary, the nondeterminism naturally present in parallel programs makes independent executions of the same program behave differently. Therefore, any mechanism that interferes with the runtime behaviour of the application will only make this effect worse by increasing the chances that different threads of control progress at different speeds. The consequence of this property is that *the results obtained by a profiling tool on a parallel execution cannot be generally used to deduce the performance characteristics of the program*. These results are applicable to the profiled

1 This reasoning would not apply if some optimisations were omitted when compiling the profiled version of the program, as explained in chapter 2.

version of the program, but this version may have very different runtime behaviour than the original one.

The core of our work consists in the design and implementation of a mechanism by which invasive profiling tools can be developed (or existing tools extended) to provide the previously itemised runtime-related information not currently accessible; while, at the same time, the parallel behaviour of the program is not compromised in a fundamental way by the interference of said profiling.

4.1.2 *Defeating nondeterminism: an event-based proposal*

From the previous discussion, we identified two problems when profiling a parallel execution. On the one hand, the requirement of minimal profiling overhead that leads to the lack of important information for an effective understanding of the performance problem. On the other hand, how the effect of nondeterminism plays a similar role than the profiling overhead, making the gathered information not truly represent the behaviour of the program being analysed.

The main techniques developed to solve this problem are two: profiling the program in a simulated runtime environment and reproducing nondeterministic executions by using execution recording techniques.

We analysed drawbacks of both approaches in previous chapters. In this section we analyse how the usage of a purely functional programming language can be used as an advantage in the implementation of the second technique.

Most of the overhead of ER comes from implementations that record all data races. In the case of a purely functional programming language, the mutability of data is restricted to thunk updates. Additionally, the absence of side effects means that the execution can be thought of as combination of both deterministic code, that performs in the same way every time, and nondeterministic events, that change from execution to execution. As an example, in our parallel Fibonacci implementation (listing 1), most of the code is deterministic (the evaluation of a simple mathematical function), and nondeterminism comes from some sparks being

garbage collected because the main thread has already updated them, and some of them being promoted to threads that can produce a result that is shared with other threads. Different executions will produce different combinations of sparks being evaluated and discarded, and threads being blocked on other threads, but the Fibonacci sequence will be computed in the same way.

Based on this fact, *we propose a new profiling method that, by tightly integrating with the execution environment of the parallel language (its runtime system), traces any nondeterministic event into a log that is later used to reproduce the same execution*. In this later *replay*, we will enable the profiling mechanism that collects the information needed to understand the behaviour of the program. Because the replayed execution is a reproduction of the original one, the information obtained by profiling it can be used to faithfully understand the original execution. A side effect of using this technique is that the profiling overhead problem is resolved because of the small amount of events that need to be traced.

4.1.3 *Execution Replay as a profiling tool*

As the main contribution of this thesis we design and implement an ER system that allows to overcome the limitations of current parallel functional profiling tools with respect to its effects on the running program and the amount of detail of the information reported. The basic mode of operation would work as follows:

1. Use the ER tool to record an execution of the program we want to analyse *without* any profiling.
2. Use the ER tool to replay the previous execution of the program *with* profiling code enabled.
3. Analyse results.

As reviewed in chapter 2, there already exist tools that allow to use ER on any program without modification, but most of them cannot be used with the described workflow. Most of them only allow to *replay*

the exact same binary that has been recorded. The replayed execution has to run in the same way as the original so that the replay log of both executions match. This means that the profiling code cannot be enabled in the replayed execution.

The record phase of an ER system needs to have very *low overhead* in order to use it in production systems. This is also a reasonable requirement for a profiling tool as discussed previously. Even though most of the reviewed ER tools comply with this, it comes at a cost: either they require hardware support, or the amount of memory-access interleaving needs to be limited. In contrast to concurrent imperative applications, parallel functional programs can work with fine-grained tasks and usually use a parallel garbage collector, which makes them the worst-case scenario for those tools.

Part of the reason that these tools have low overhead is because they do not maintain any *coordination between threads* except when the nondeterministic execution requires it (inter-thread communication). That is, the recording log that is saved in the first execution can be viewed as a collection of independent logs, one for each thread ever created by the program, that are only related to each other though some nondeterministic events. This seems to be a good trade-off that should not harm the ability to fix a bug: the replayed execution may show an unsynchronised state between threads at a particular point in time, but each thread will have the same view of the environment it originally had so that the debugger can trust any value affecting the thread's behaviour. On the contrary, if we are to use an ER system to debug a performance problem related to parallelism, it is extremely important that, when inspecting the state of the program, all threads reflect the original evaluation progress as precisely as possible.

Our goal is to design an ER tool that is able to comply with the requirements stated in section 4.1, while, at the same time, avoids the problems listed above. These requirements can be summarised in the need for very tight integration between the profiler and the runtime environment. For this reason, we plan to implement it as a runtime system extension that communicates its results via a tracing mechanism. At the same time, we take advantage of two facts made available for our use case (perform-

ance profiling of a lazy purely functional language). First, because the type system guarantees the absence of side-effects, the amount of nondeterministic events is greatly reduced. Second, we are debugging the performance of the program, so our replay only needs to reproduce the same interactions between threads, but can omit other details of the execution *such as requiring the same code to be run. Therefore, as long as the program's code remains the same*, the implementation of the profiler, that is contained in the runtime system, can inspect the program's runtime representation and gather and trace any information about it.

The replay of an execution would interleave nondeterministic and deterministic events maintaining the ordering of the nondeterministic events. While the exact state of the system would not be preserved between nondeterministic events, the information related to the parallel behaviour of the application would be relevant as long as those nondeterministic events were interleaved in the same way.

Additional use cases are enabled by our approach:

- 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 the granularity of the threads created from sparks, in others, we might be interested in discovering what data is being garbage collected. With our ER design, there exists the possibility of dynamically adjusting the type and level of profiling detail during 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 ER, 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 as in the original run.

4.2 EXECUTION REPLAY IN GHC

We have chosen as implementation target GHC. Our work was implemented by defining a new replay *way* (see section 3.1.2) that enables the logging of all nondeterminism to an event log file and the reading of a file built like that to replay a previous execution. We first give an overview of how the implementation of our new profiling mechanism into this platform was tackled and then proceed to expand on the details about the recording phase (section 4.2.2) and on the sequential and parallel replay (section 4.2.3).

4.2.1 *Implementation overview*

We discussed in section 4.1.1 that any program usually involves both deterministic and nondeterministic behaviour. Because we are restricting our tool to a purely functional programming language, there is a guarantee that the program's behaviour is going to be deterministic. Then, any nondeterministic behaviour is going to be hidden in the runtime system to deal with functionality external to the program semantics, mainly, to deal with parallelism and garbage collection. Our strategy is to expose any such nondeterministic behaviour as events emitted by the event logging mechanism available in GHC's RTS (see section 3.1.2). When the program is launched in replay mode, *all* it has to do is read the previously generated event log and run the program making sure that those events happen in the same way, that is, ensuring the same ordering and providing the same information.

Capabilities are the abstraction that allows a Haskell thread to execute in the way we described in section 3.1.2. So, from all tasks available in the RTS, only the ones currently owning a capability will be running and emitting events. A few events will be global, not specific to any capability, while most of them will be capability events, emitted within a single capability and concurrently with other events. When replaying, the program will spawn a *replay thread* whose task is to run a loop which reads the next event to be emitted, discovers which task was responsible

for the event, and instructs the task to run just until that event is emitted, as shown in pseudocode (listing 6). Obviously, while some events will not require any extra actions in order to be reproduced, most of them require some setup either before or after the task is run. This will be discussed later in sections 4.2.2 and 4.2.3.

Listing 6: Pseudocode for the replay thread

```
void replayLoop (void) {
    while (1) {
        ev = nextEvent();
        setupNextEvent(ev->cap);

        task = eventTask(ev->cap->no, ev);
        signal(task);
        wait(replay_thread);

        if (ev == finished)
            break;
    }
}
```

Each task, when emitting their corresponding event, will call a function `replayEvent(ev)` which checks that the event emitted matches the one read from the event log:

Listing 7: Pseudocode to replay an event

```
void replayEvent (ev) {
    read = readEvent();

    if (ev != read)
        exit(error);

    switch (ev->type) {
        ...
        case 'event n':
            setupEvent(ev);
        ...
    }
```

```
13
14      signal(replay_thread);
15      wait(task);
16  }
```

After checking that the events are equal (line 4), modulo its timestamp, depending on the event type there may be some bookkeeping to do (lines 7–12): setting the environment so that the program arguments match the original ones, storing a new task id for a new task, etc. Those cases will be analysed in depth in section 4.2.3.

An excerpt from the textual representation of the trace of a `pfib` execution described in section 1.1.2 is shown below:

Listing 8: Excerpt from `pfib`'s event log

```
1  ...
2  4177926000: cap 1: stopping thread 4 (stack overflow) (96 words
       allocated)
3  4177940000: cap 1: running thread 4
4  4180949000: cap 1: stopping thread 4 (heap overflow) (65024
       words allocated)
5  4180979000: cap 0: stopping thread 3 (blocked on blackhole owned
        by thread 4) (25253 words allocated)
6  4181027000: cap 0: task 1 releasing
7  4181146000: cap 1: running thread 4
8  ...
```

In this example, the first three events would not require any preparation. `thread 4` would stop after finishing its stack (line 2), then the replay thread would find out that the next event happens in the same capability, so it chooses the same task to run. `thread 4` would start running again (line 3) and stop with a heap overflow (line 4), which, again, needs no setup because the nursery size should match the original one as configured at startup. The next event, *stop thread* corresponds to a different capability (line 5). The function `eventTask()` will return the task owning `cap 0` at that point. Additionally, some preparation needs to be performed: the thunk on which it will block must already be a blackhole, otherwise `thread 3` will enter it and start running the thunk's code. As

will be shown later, this is done by checking in the entry code whether the thunk should have been evaluated or not. In the first case, control would be given to the thread that evaluates it until it updates the thunk. In the second case, the thread would just evaluate the thunk.

4.2.2 *Recording phase*

In section 3 we described how event logging works in GHC [20] and some of the events that are typically logged. For our purposes, we had to identify already available nondeterministic events to preserve in the event log and add new ones that were not present. We have classified the events depending on the purpose of the information they carry. Some of the events are not explicitly needed to replay but are important because of the information they represent and they are kept in the description. Required events are marked with an asterisk to differentiate them from the rest and, from those, newly-added or modified events are marked with a dagger.

GENERAL EVENTS From the start of the program there are some basic events already present in GHC that help setting things up. When starting up, there is a *startup* event that indicates the amount of capabilities configured to be run. We reuse this event to emit it at the end of the program, with zero capabilities to indicate the last received event. After this one, the RTS emits a few events related to system configuration: *rts identifier** that encodes the ways (see section 3.1.2) built into the RTS, *program args** and *program env** that contain strings with the arguments and all environment variables used when running the program. These events are used to check that the replayed program was built with the correct RTS configuration for replay (the replay RTS way) and to recreate the same environment and options passed originally. This allows us to replay a program using a simplified command line:

```
$ ./program +RTS --replay
```

CONCURRENCY EVENTS As described earlier, capabilities are the main concurrency unit of the runtime system as they limit the amount of operating system threads running at the same time. Tasks, the runtime system representation of the real OS threads, need to acquire a capability in order to run Haskell code.

The task lifecycle is managed by the existing events *task create**, *task delete** and *task migrate**. These events record the ids of the tasks involved and the destination capability number in the last case.

Capabilities are handed over from running tasks when they yield its control to other tasks that were previously waiting. This behaviour is recorded by using three new events: *task acquires capability*†, *task releases capability*† and *task returns to capability*†. The last one is emitted when a task is ready to run Haskell code, so it is enqueued in that capability until it is available. It is used, for example, when returning from a foreign call or to start an in-call (e.g. the Haskell thread running `main` that is called from the RTS). These events store the task ids and the number of the capability they act on.

Capabilities change its owner task when a task sets the capability variable `running_task` to either `NULL` or to a `Task` pointer, freeing it or acquiring it, respectively. Because the amount of capabilities is restricted and tasks run concurrently, to exchange control of a capability between two tasks, a lock is used to avoid a race condition. In order to ensure the proper ordering of the corresponding events, we make sure that the lock is not released until the events are emitted.

Lastly, message communication between capabilities is handled with the new events *send message*† and *process inbox*†. Capabilities send messages to each other concurrently, and each of them checks their message inbox at any time. In this case, we do not need to assign ids to every message and log each message being processed. This is because when a capability sends a message, it first needs to acquire exclusive access to the destination capability so the order of sent messages is preserved. At the same time, a capability checking its inbox is also going to do it atomically. Those two facts together ensure that the timestamps of every message being sent are going to be smaller that the timestamp of the

event for the inbox processing of those events, and any message not checked will have a bigger timestamp.

THREAD SCHEDULING EVENTS A description of the threading system in GHC was already presented in section 3.1.2. As described there, logical Haskell threads executed on the same capability are scheduled in a round-robin fashion. There are three main causes for a thread to stop running and return control to the scheduler:

- the thread runs out of heap or stack space (in the first case garbage collection needs to be performed before any thread can continue running);
- the thread blocks on a blackhole; or
- its time slice expires.

In all of the above cases, the thread is preempted and the next thread in the queue is selected for execution. The thread lifecycle is tracked by using the following events: *create thread**, *run thread**, *stop thread**, *migrate thread** and *thread wakeup**. All of these events contain the thread id and, in the case of *migrate thread*, the destination capability. While the rest are self-descriptive, we should clarify the usage of *thread wakeup** and *stop thread**. The former, is the event issued whenever a blocked thread is released and enqueued in a scheduler run queue. It happens, for example, right after a blackhole is updated or a thread is migrated. This event is also needed to ensure proper ordering in the scheduler run queue. *stop thread** contains an additional parameter: thread status, that can be one of `HeapOverflow`, `StackOverflow`, `ThreadYielding`, `ThreadFinished`, `ForeignCall` and a number of `BlockedOn..` statuses (`BlockedOnMVar`, `BlockedOnBlackHole`, etc.).

Of special interest is the stop event whose status is blocked, in the case of blocking on a blackhole, because then it also stores the thread id of the owner of the blackhole. Similarly, `ThreadYielding`, that indicates that the thread did not forcefully stopped. In these two cases, the thread is stopped nondeterministically, either by blocking on a thunk being

evaluated by a thread running on another capability, or by an external clock expiring the thread's time slice. Additionally, we have added a new *capability allocation*† event that is emitted when the thread yields, and stores the amount of memory allocated by the thread up to that point. As the name implies, that allocation is accounted per capability. In section 4.2.3, we will see how this information is obtained and used to force the thread to stop at the same point in the replayed execution.

When a thread stops and returns control to the scheduler, there is the option of appending it to the end of the run queue or enqueuing it at the beginning. This action depends on the event that led to the thread stopping. The first case happens when the thread finishes its time slice or when forking new threads (this is done to improve responsiveness). In any case, there is the need to identify this event happening and act upon it. For that, instead of creating new events for each case, we emit just a new one every time a stopped thread is added to the run queue indicating whether the thread switched context; that is, whether it was replaced by a different thread. This *context switch*† event allows us to insert the thread in the right position of the run queue at replay time.

Another related event is *create spark thread**. This event is emitted when the scheduler has no more threads to run but there are sparks available in some capabilities and a spark thread is created. This thread is in charge of finding runnable sparks and evaluate them. The nondeterminism of this event origins in the fact that the check to activate the spark thread has to look for sparks in all capabilities.

Besides events related to a thread lifecycle, we had to add some events that were used to track the scheduler state and allowed to identify when a scheduler is finished. `sched_state` is a global variable that is used to shut down the system and also to run the last compulsory garbage collection before shutting down. This is handled with its three possible values: `SCHED_RUNNING`, `SCHED_INTERRUPTING` and `SCHED_SHUTTING_DOWN`. Because the behaviour of a the scheduler is going to be dependent on this value, we emit a *scheduler state*† event each time it is read. Additionally, another new event *scheduler finished*† is emitted to track when to exit the scheduler.

PARALLELISM EVENTS Some of the more important events are the ones related to parallelism. In the parallel model of GpH, these are the events related to thunk updates, in particular to shared thunks. A requirement for being able to replay a program correctly is that we are able to identify shared expressions and which threads evaluated them. We will need to log when threads are blocked on blackholes, and when they obtain the thunk's result, computed by a different thread. Because of this, thunks need a way to be identified from when they are first entered until they are updated with the WHNF result. Then, after the next GC, the blackhole will be removed and we are no longer interested in tracking it.

To trace the information related to thunk updates we need a way to generate and assign identifiers to shared thunks. There are two dimensions to consider:

1. The process must be reproducible so that the same ids are used for the same expressions in both the original and replayed execution.
2. There is an indeterminate amount of expressions that can be eventually shared. The cost of both identifying and tracing the information must be minimised.

First, a place to store ids is needed. We discarded adding an additional field to `StgThunk` (listing 4) which would increase binary sizes and reduce performance for every thunk, even if not shared. Fortunately, thunks already have a word-sized free space available for storing its result. The scheme we used combines the usage of an integer id that is saved in the thunk itself, and the thunk pointer once it is updated.

Using the thunk pointer requires to trace pointer changes at each GC. On the contrary, while the thunk has not yet been updated, its id is kept unmodified and no action is needed. To extend the lifetime of the id, we changed the way in which blackholing is performed. Instead of overwriting the thunk's result with the thread pointer, we keep the thunk id but reserve part of the word space to store the thread id. Then, in situations where the thread pointer is needed (when blocking on the thread or copying the closure in GC) we recover the thread from an array mapping

thread ids to the thread itself. This encoding allows to reduce the usage of pointers as identifiers as much as possible at the cost of limiting the amount of threads and thunks available in a program execution. Then, only after a thread is blocked on the thunk (and a blocking queue must be saved in the thunk's result) or the thunk is updated, the id will be overwritten.

Once we start mixing pointers and ids in the same space (the thunk's result space), we need a way to tell one from the other. In order to do that we decided to add another limitation by imposing the usage of 64-bit platforms. All currently available 64-bit architectures use 48 bits for virtual addresses. This allows the use of the 16 more significant bits to mark pointers as either ids or real pointers.

Identifier creation is handled when allocating thunks. At that point, a new natural number is created by increasing the last id used in the capability (this new id is stored there). In the case of CAFs[2] [108], their id is built from the static code address they point to, a value that is guaranteed to be constant and to fit the available space. For dynamic thunks, the id of the capability in which they are allocated is encoded in the thunk id by using the following function:

Listing 9: Function to generate a new thunk id in a given capability

```
1 nat newThunkId (Capability *cap) {
2     nat capno = cap->no;
3     cap->thunk_id++;
4     return cap->thunk_id + (capno+1) << THUNK_ID_BITS;
5 }
```

where `THUNK_ID_BITS` further splits the id space between capability number and thunk id. This encoding allows to use `newThunkId()` from different capabilities without using locks and, at the same time, it can be used during replay to obtain the capability in which a thunk was allocated. The capability number can later be retrieved by shifting the id:

Listing 10: Function to obtain the capability a thunk was allocated on

2 Constant applicative forms (CAFs) are top-level thunks, defined as part of the static data of the program executable.

```
1 nat capThunkId (nat id) {
2     return (id >> THUNK_ID_BITS)-1;
3 }
```

Capability numbers are shifted by one unit so that ids from CAFs do not clash with the ones from dynamically-allocated thunks.

As mentioned earlier, the 16 most significant bits are used to distinguish ids from pointers. We distinguish the different kind of ids with the usage of the corresponding atoms listed in table 1.

Table 1: List of atoms used in addresses to identify thunks

Atom	Meaning
REPLAY_ID	Initial value used for thunk ids
REPLAY_TSO	Thunk has been blackholed
REPLAY_SHARED_TSO	Shared thunk has been blackholed
REPLAY_PTR	Shared thunk points to result
REPLAY_SPARK	Sparked thunk

REPLAY_ID is used initially to assign an id to a thunk. The rest of the address contains the thunk id. REPLAY_TSO and REPLAY_SHARED_TSO are used in place of blackholing. As said earlier, instead of overwriting the thunk's result with the thread pointer, we keep its initial value, add the thread id and overwrite the marker indicating its new status. The current implementation uses 16 bits for thread ids and 32 bits for thunk ids. REPLAY_SHARED_TSO is used when a thunk is shared and a thread running in a different capability from the one allocating it blackholes the thunk. REPLAY_TSO is used in the general case that a thread blackholes a thunk allocated in the capability in which the thread is running. When a thunk is finally updated or needs to point to a blocking queue to hold blocking threads, it will be updated with a pointer. In this case, if the thunk was shared (REPLAY_SHARED_TSO), the pointer will be tagged with REPLAY_PTR; otherwise, the thunk loses its tag. REPLAY_SPARK will be used

for sparks, so that later, when a spark is fizzled or garbage collected, it can be recognised from its pointer.

Even though sparks are the initial source of parallelism (given the GpH parallel model), many more thunks will be shared during the execution of a program. There are three situations in which a thunk is accessible to more than one thread:

1. CAFs are available to any thread from the start of the execution. They represent global expressions not allocated at runtime but defined in the program binary and they are already referenced from anywhere in the program.

2. The spark payload. Any free variables that appear in an expression which is sparked become accessible to a thread stealing the spark (besides the one that sparked the expression).

3. Thunks reachable when a shared thunk gets updated. Whenever a previously shared thunk (stolen spark or CAF) is updated with an expression not in normal form, all the new thunks that are part of the updated value are then accessible to the thread that originally allocated the initial thunk; or to every thread, in the case of CAFs.

Because of this (especially the third point), the amount of sharing present in the most simple parallel program can be enormous. A naive way of tracking nondeterministic events related to thunk updates would be to emit an *enter thunk* event each time a shared thunk is entered, tying together the thread and thunk ids. To identify shared thunks we would assign new ids to each spark created and also in each of the three cases presented above. That is, we would assign ids to all CAFs and we would traverse the thunk's payload recursively assigning new ids in the case of a spark being created or an already shared thunk being updated. As a matter of fact, we tried to do this initially, but found the overhead of the traversal to be too large. The problem manifests when the sparked expressions contain big data structures, as it happens with many parallel algorithms (e.g. sorting). In those cases, a common parallel strategy is to split the input data in pieces and work with them in parallel. Even

if a thread fully evaluates its data, there is no way of knowing that, and when a shared thunk is updated, we need to traverse the shared value again, in case it finds an unevaluated thunk to assign an id to. Even if we could find a way to assign the ids when shared thunks were created, and therefore avoid the new traversal later, a worse problem would still remain: the program would emit an event for each shared thunk, even in the case of thunks that become visible to other threads but are never actually used by them.

The more lightweight design that we used to track parallel updates relies in the fact that most thunks in the program execution will be updated by the thread that allocated them. When an expression is sparked and evaluated by a parallel thread, the evaluation of the expression will result in the creation of many intermediate thunks but, eventually, either evaluated or discarded, and the expression reduced to a WHNF value and stored as the thunk's result. Most of those intermediate thunks allocated by the parallel thread will not become visible to other threads. Given this, our design involves the creation of ids at thunk allocation time for every thunk in the program, as described earlier. Then, when the thunk is evaluated, a new *enter thunk*† event is only emitted if the thread entering the thunk is running in a different capability than the one of the thread that allocated it or if the thunk is tagged with `REPLAY_SPARK`. By doing it in this way, we fix the two problems described earlier: no additional traversal is needed because thunk ids are pre-assigned at allocation time, and the amount of events is greatly reduced by emitting them only for thunks evaluated by a parallel thread. Checking for a different capability instead of a different thread works in a similar way but may afford to trace some events where the thread entering the thunk may have migrated from another capability to the one in which the thunk was allocated.

While the described design allows to track which shared thunks were entered in a lightweight way, it suffers from one drawback. If two threads enter the same thunk before it is blackholed, it can occur that also both of them update it without detecting a collision or suspending the computation. Being a purely functional runtime, we can replay the program and choose any of the results as the last to be written, and the program's

result would remain unchanged. But it can happen that between those two writes another thread reads the thunk's result and further evaluates it. In this case, during replay, we would fail if we do not reproduce the exact updates and reads in the same order. While many programs may work, in particular numerical algorithms where sparks are evaluated to normal form such as integer results, this scenario may not be uncommon for fine-grained parallelism. To solve this, we recommend the usage of eager blackholing, that immediately blackholes a thunk when it is entered and, even in the case that two threads happen to enter at the exact same time, also resolves duplicated computations at the time of update. Doing this, the only chance for this data race to happen would be a thread getting descheduled just after reading the thunk's header and scheduled back after the other thread already updated the thunk, so that the thunk is blackholed again by the returning thread. We believe the chances for this to happen are extremely low and, in any case, we discuss some possible solutions in chapter 6. In the rest of the section, the description will assume that eager blackholing is enabled.

The *enter thunk*† event carries the thunk id and pointer as parameters, allowing at replay time to relate both of them. At the time of entering a blackhole, we discussed in section 3.1.2 three possibilities, and each of which has a matching event: *blackhole WHNF*†, in which the blackhole is evaluated to a WHNF value; *blackhole message*†, in which the blackhole still points to the thread that is performing its evaluation and the current thread will block on it by sending a message to the owner capability; and *blackhole thunk*†, when the blackhole points to another thunk. Of those, only *blackhole message* is always emitted and because it may still have the thunk id available and it will log, in the same way as *enter thunk*, both the thunk id and pointer. If the blocking thread is not the first, the blackhole will be pointing to a blocking queue, and it will only log the thunk pointer. In the first case, the id is needed for when an event was not emitted when entering the thunk. The other two events are only emitted for blackholes tagged with `REPLAY_PTR` and, because they only have access to the pointer, that is the information that is recorded.

We described in section 3.1.2 that some threads may not finish the evaluation of a thunk and instead suspend it if they find other threads

doing the same thing when they are paused. The new event *suspend computation*† uses the thunk pointer to identify the thread that will cancel the evaluation of the thunk. Similarly, collisions when finally updating the thunk are logged with *collision WHNF*† and *collision other*† that are used to distinguish both collision cases previously described. These two events also use the thunk pointer. In the second case, when the collision happens with a blackholed thunk, the thread also discards its result in the recording phase, so that we do not introduce a nondeterministic update if the other thread happens to update it at the same time.

Finally, a new *pointer move*† event is added to keep track of thunk pointers between each GC until they are updated. Only blackholes pointing to blocking queues need to be logged, because, when there are no threads blocked, the blackhole still has the thunk id and, when the blackhole is updated with the final result, we can discard the tag at the next GC. The garbage collector is in charge of resetting any updated thunk (actually removing the tag from the indirectee pointer) when the blackhole is discarded.

Besides thunks, the spark lifecycle also needs to be logged in order to coordinate between sparks being created and stolen from different threads. Spark events are related to what happens when they are created. An *spark created*† is initially emitted. Then, the spark is inspected to check if it is already evaluated, emitting an *spark dud*† event in that case. When adding the spark to the spark queue, it can either overflow, recorded with a *spark overflowed*† event, or be correctly saved, emitting a *spark converted*†. Then, when sparks queues are inspected looking for work, evaluated sparks will result in a *spark fizzled*† event, and a *spark garbage collected*† event will be emitted if the thunk did not survive a GC. If successful, a spark run from the same capability in which the thread is running results in a *run spark*† event and, if it is instead stolen from other capability, a *steal spark*† event is used. All of these events were modified from existing events to carry thunk ids except for a stolen spark that also has the id of the capability it is stolen from and a spark being fizzled or garbage collected. In these two last cases, the thunk has already been updated so that the id is not available, and they use the thunk pointer. The usage of the `REPLAY_SPARK` atom forces the *enter thunk* event to happen so

that, at replay time, the thunk pointer used by these two events will be recognised.

GARBAGE COLLECTION EVENTS There are many events related to GC. *GC start*, *GC end*, *request sequential GC**, *request parallel GC**, *GC idle*, *GC working*, *GC done* are emitted all through the GC process and they are mainly used to represent it in capability profiles.

Additionally, there exist some events used for statistics reporting. Those are *heap allocated*, *heap size*, *heap live* and *GC stats* to report runtime allocation data, and *heap info* to report the collector configuration.

As we will see later, at replay time, GC is performed sequentially so that it does not need to be traced in detail. We only need enough information to coordinate the different phases of GC: the collection itself, and cleaning the spark queue as described in section 3.1.2. *request sequential/-parallel GC* identifies the capability in which the collection is requested (and therefore, the task in charge of coordinating it), *end GC*† marks the point from which all alive data has been identified and is emitted by the main GC task and *prune spark queue*† is emitted by all GC threads after finishing cleaning the spark queue.

The following table compiles the events that are required for ER and the information they carry:

Table 2: List of required events and the information they provide

Event	Information
RTS identifier	Encodes the runtime system configuration
program args	Program arguments
program env	Environment variables
create thread	Id of the new thread
run thread	Id the thread about to run
stop thread	Id of the stopped thread

Event	Information
migrate thread	Id of the thread being migrated and the destination capability
thread wakeup	Id of the thread being woken up
create spark thread	Id of the new spark thread
capability allocation	Allocation at which a thread yielded
context switch	Whether the thread context switched with the next one in the run queue
scheduler state	Global scheduler state that tracks the shutting down sequence
scheduler finished	Emitted when the scheduler is finished
spark created	Id of the new spark
spark dud	Id of a new spark already evaluated
spark overflowed	Id of a spark that overflows the spark queue
spark converted	Id of a spark added to the spark queue
spark fizzled	Id of a spark already evaluated
spark garbage collected	Id of a spark garbage collected
run spark	Id a spark taken from the thread's capability
steal spark	Id of a spark being stolen and the capability id
enter thunk	Id and pointer of the thunk being evaluated
blackhole WHNF	Id of the blackhole whose result is returned
blackhole message	Id and pointer of the blackhole where a thread gets blocked
blackhole thunk	Id of the blackhole whose result is entered
collision WHNF	Pointer of the blackhole whose result is reused
collision other	Pointer of the blackhole which is not yet updated

Event	Information
suspend computation	Pointer of the thunk whose evaluation is cancelled
pointer move	Old and new thunk pointer after GC
task create	Id of the new task
task delete	Id of the deleted task
task migrate	Ids of the task migrating and the destination capability
task acquires capability	Id of the task that acquired the capability
task releases capability	Id of the task that released the capability
task returns to capability	Ids of the task and capability in which it waits
send message	Id of the destination capability
process inbox	The message inbox was checked
request sequential GC	A sequential GC was requested
request parallel GC	A parallel GC was requested
end GC	The GC traversal is finished
prune spark queue	The spark queue was cleaned

4.2.3 *Replay phase*

The overview from section 4.2.1 explained how the main task of the *replay thread* is to act as a scheduler that interleaves the execution of the different tasks to reproduce the events read from the event log that was saved in a previous execution. The replayed event log is read into per-capability buffers (and an additional buffer for global events) and `nextEvent()`, shown in listing 6, compares the timestamps of the first event in each buffer to return the one that happened earlier. Using separate buffers for each capability allow to inspect what happens next in each capability independently.

An important part of the scheduler consists on modifying the environment in which those tasks are run so that they behave in a way that the same nondeterministic events emitted in the original run are emitted again in the same way. We will now classify the events into those required for sequential replay and those required for parallel replay, and describe what actions are needed for every event described in table 2.

Sequential replay

By sequential replay, we refer to the replay of programs where the Haskell code runs on a single processing unit. That does not mean that there will be just one thread running in the system, but that the execution will be limited to one thread at a time, interleaved with others in one run queue from just one capability. Additionally, there can be more than one task (real OS threads) running, although only one will be the owner of the capability and thus responsible for scheduling threads.

INITIAL SETUP The *program args* and *program env* events are the first events emitted at the beginning of the program. They are emitted after the RTS does the argument parsing to configure itself. When replaying, the parsing happens in the same way and, just after that, the replay system is initialised and those events are read from the event log so that we can call the RTS setup again (`setupRtsFlags()`), now with the saved arguments and environment variables from the first execution, and also copy the original environment.

The event *RTS identifier* is just used to check that the RTS was built with the same subsystems enabled in both executions, and that the `replay` RTS way was enabled so that the events needed for replay are available.

CONCURRENCY The replay scheduler sets a semaphore for each task in `task_replay[]` that is created in order to coordinate them, and a single `no_task` semaphore is used for code run before the first task is available. The replay loop decides which task is the next to run depending on the event just read. Because there is only one task running at a time per capability, we keep a `running_tasks[]` array indexed by capability num-

ber that tracks the active task. Given a per-capability event, finding out which task needs to run next is just a matter of checking `running_tasks[]`, as long as we keep it updated when tasks yield control from its capability and other tasks take over. There are a few exceptions to this: when a *task acquires capability* event is read, we have not yet assigned the task to the capability; when a task is deleted and hence it relinquished its capability first; and for global events that have no capability associated. We can check the code for `eventTask()` in listing 11:

Listing 11: Pseudocode to select the next task to run.

```
int eventTask(nat capno, Event *ev) {
    int taskid = -1;
    switch (ev) {
    case EVENT_TASK_DELETE:
    case EVENT_TASK_ACQUIRE_CAP:
    case EVENT_TASK_RETURN_CAP:
        taskid = ev->task;
        break
    default:
        if (capno != -1) {
            taskid = running_tasks[capno];
        }
    }

    if (taskid == -1 && replay_main_task != -1) {
        if (hs_init_count == 0) {
            taskid = replay_main_task;
        } else {
            taskid = replay_main_gc_task;
        }
    }
    return taskid;
}
```

Fortunately, the cases involving tasks do not need special measures because their events carry the task id we need. There are two cases that need to be handled especially: initialisation and shutdown. When the runtime system starts (`hs_init_ghc()`), there are a few global events that

are not run by any task, because the task subsystem is not yet initialised and we have not even run any Haskell code yet. To replay those events (the ones described in section 4.2.3), instead of using the per-task semaphore we use the `no_task` semaphore, that is signalled every time `eventTask()` returns -1. When shutting down, the last events are also global but the only task alive at the time is the one that also initialised the system. At startup, we identify the initial OS thread storing its thread id in `replay_init_thread` so that, later on, each time as task is created, we check if its OS thread corresponds to that one and save the task as `replay_main_task`. `eventTask()` knows that the RTS is starting up when the main task has not been defined yet (line 15) or that it is shutting down with `hs_init_count` (line 16), which is set to zero after the main thread finished evaluating `main`.

There are two types of tasks: worker tasks and bound tasks (section 3.1.2). When a worker task is created, the call chain works like this:

Listing 12: Call chain when starting a new worker task

```
1 startWorkerTask() [parent]
2     newTask()
3     createOSThread()
4         workerStart() [child]
5     traceTaskCreate()
```

In the case of a bound task, `newBoundTask()` will not create a new OS thread, it will make the current thread bound to the new task. Additionally, it will only create a new `Task` structure and emit the *task create* event if it is the first *incall* to Haskell, otherwise, it will reuse the current task.

In both cases, we modified the function `newTask()`, so that after the `Task` structure is initialised we call `replayNewTask()` to add a new entry in the `task_replay[]` array initialising a semaphore for the new task. The OS thread associated with the new task is spawned with `createOSThread()` to run the function `workerStart()`. This function will perform the needed setup and eventually call `scheduleWorker()` where this task will start running threads from its run queue. Before doing that, the task calls `replayWorkerStart()` so that the newly created task is stopped, waiting on the previously created semaphore. A new task is always associated

with a free capability, so that it can immediately start running. That means that the first event it emits is *task acquires capability*, allowing the replayer to identify the task to run after the current event. Placing the task initialisation (therefore the setup performed for replay too) and the emission of the new task event in the parent task, makes it easier to coordinate the replay. If the event was emitted by the new task, our replayer would have to identify the task *parent* and instruct it to run in order to create the new task and forcefully stop it. The current setup only places an *spurious* wait just after a new OS thread has been created as explained above.

Capability ownership is handled by the set of events *task acquires capability* and *task releases capability*. When they occur we save or reset the owner task in `running_tasks[]`. Apart from a new task being created and its capability being handed over directly, acquiring a capability happens in a variety of contexts: when a thread returns from a foreign call to continue its execution, to initiate a garbage collection in a situation that does not involve a heap overflow (where the task already owns a capability); or when all capabilities need to be synchronised (in that case, one task is in charge of acquiring all capabilities). In all those cases, the task waiting for a capability will call `waitForReturnCapability()`, that will acquire a free capability or enqueue the task in a *returning tasks* list belonging to a non-free capability until the capability becomes available and the task is signalled to acquire it. Another case is when a task explicitly yields control of the capability with `yieldCapability()` and waits in a way similar to that of a returning task for the capability to be free again. Acquiring the capability occurs implicitly by storing the task as the capability's `running_task`. We emit the *task acquires capability* event just after the assignment. This action is protected by a lock and we explain later how this fact is used to order the capability ownership events.

A task releases control of a capability by using the function `releaseCapability_()`. It works by resetting the capability's `running_task` and looking for a returning task to give the capability to, eventually leaving it free if there are no returning tasks, nor a bound task on top of the run queue, nor any spare workers. The other event related to tasks is *task*

returns to capability. In the case of sequential replay, this events do not require any further actions.

THREAD SCHEDULING We saw that thread execution is traced by using the following events: *create thread*, *run thread*, *stop thread*. Additionally, *capability allocation* is always emitted before *stop thread*.

While the first events are mainly used for the information they contain, they will happen deterministically: threads will be created and run independently of any external event. In opposition to that, for *stop thread* to happen, some conditions need to occur. There are two types of conditions: threads are forcefully stopped when trying to evaluate a blackhole, when they are finished, when making a foreign call, or when the heap and/or the stack have been exhausted. By forcefully, we mean that the thread is immediately stopped at that point. The other case, that corresponds with the status `ThreadYielding`, happens when the thread is either asked to stop to give other threads the opportunity to run or the RTS timer ticked and the thread's time slice is finished. In the first case, only the evaluation of a blackhole is a nondeterministic event that interacts with other threads in the system, and will be explained later. In the second case, we need a way to identify when the thread stopped.

We cannot rely on a time measure to act as a proxy for the number of instructions executed because it would lack the needed accuracy and latency (the overhead of calling an OS clock function through several software layers and the overhead of the machinery to stop execution at a very precise time) and the accuracy and hardware variations of instruction counters make them not reliable either [109, 110]. Fortunately, in GHC's RTS, a thread can only stop at safe points and, when a thread needs to yield, that safe point is always the stack and heap check: the moment in which the nursery is inspected to check if there is enough memory to perform the next allocation.

This mechanism gives us an easy way to record the point at which a thread stopped. We just need to calculate how much memory has been allocated by that thread until it stopped. Thus, an event *capability allocation* is always emitted before a *stop thread* event with the needed information. For this purpose, we introduced a pair of functions `replaySaveHp()`

and `replaySaveAlloc()` that are called just before a thread runs and when a thread is paused, respectively. Their task is to save in the capability the current value of `Hp` and the block pointer in the first case, and account the amount of memory allocated since the last call in the second case. The `Hp` pointer alone is not enough because it points to the last written byte in the block so that it will have the same value for a full block and an empty block that is contiguous in memory to that one. Because of this, `replaySaveAlloc()` counts both the amount of allocated blocks (only full blocks) and the amount of allocated bytes.

When replaying, the RTS timer is disabled and we need to calculate the point in which a thread is going to stop before the *stop thread* event happens. The replay loop in listing 6 contains a call to `setupNextEvent()` that is in charge of doing preparation work for the next event. In this case, it checks if the next event is a *stop thread* event and it reads the next event in the same capability, that will be a *capability allocation* event, to calculate the future heap and block pointers in which the thread has to stop. The source code looks like this:

Listing 13: Pseudocode for settings the thread's heap allocation limit

```
1  void setupNextEvent(Capability *cap) {
2      CapEvent *ce = readEvent();
3
4      if (ce == EVENT_STOP_THREAD &&
5          ce->status == ThreadYielding) {
6          ce = peekEventCap(1, ce->capno);
7          ASSERT(ce == EVENT_CAP_ALLOC);
8          int alloc = ev->alloc;
9          int blocks = ev->blocks;
10         bdescr *bd = cap->replay.last_bd;
11         void   *hp = cap->replay.last_hp;
12
13         // add the allocated memory from the first block, so
14         // the loop counts full blocks in every iteration
15         alloc += hp - (START(bd) - 1);
16         while (blocks > 0) {
17             alloc -= SIZE(bd);
18             blocks--;
```

```
19              bd = bd->link;
20          }
21
22          cap->replay.bd = bd;
23          cap->replay.hp = (START(bd) - 1) + alloc;
24      }
25  }
```

As one can see, we save the target `Hp` and block pointers in a `replay` structure stored in the capability (lines 22–23). Then, during replay, when a thread starts running and sets up its `Hp` and `HpLim` pointers, we load the latter from `cap->replay.hp`. Later, the code that checks `HpLim == NULL` to yield the thread will also check whether `Hp` is equal to `cap->replay.hp`, and likewise for the block pointer.

After a thread is paused, the *context switch* event is used to decide whether to put it at the front or back of the run queue.

Parallel replay

The implementation of replay for parallel programs, even if much more complex, follows the same model described for sequential replay. Events need to be replayed in chronological order, and to simplify its implementation, we replay the execution sequentially. This means that, even if the program uses multiple threads concurrently, the replayed execution will interleave them so that only one thread is running at a time.

CONCURRENCY Task-level concurrency is handled in the same way as described in section 4.2.3. There is a difference, however: in the sequential case, the lack of concurrency meant that *task releases capability* events always happen before *task acquires capability* events. This makes it easy to replay both events: we track the first task yielding the capability and the second one acquiring it by updating the corresponding value in the `running_tasks[]` array. In the parallel case, concurrent execution means that, if we are not careful, those events can be emitted in the wrong order. Acquiring and releasing a capability is an operation protected by a lock. Most of the times, the capability is released using one of the following set of functions: `releaseCapability()`, `releaseAndWakeupCapability()` and

`releaseCapabilityAndQueueWorker()` that lock the capability before releasing it. Besides doing other work, all of them follow the same pattern illustrated by the source code of `releaseCapability()`:

```
void releaseCapability(Capability *cap) {
    ACQUIRE_LOCK(&cap->lock);
    releaseCapability_(cap, rtsFalse);
    RELEASE_LOCK(&cap->lock);
}
```

In order to correctly replay the interleaving of these events, we make use of the locking that is already in place to coordinate capability ownership and make sure that, when releasing a capability, a task does not release its lock until after emitting the event and, in contrast, the event emitted to acquire the capability happens after taking the lock to do so.

For the same reason depicted above, *send message* and *process inbox* events need the same kind of coordination. Again, the existing locking mechanism that protects inbox access is used for this purpose but in a more restricted way. The difference here is that sending a message does not impose any requirement on the inbox (like ensuring it is empty or so) in the same way that a task needs the capability to be free in order to acquire it. Then, to prevent the absence or addition of events for messages that were or were not checked, respectively, in a capability's inbox, both events need to be emitted inside the locked area.

Other events, also related to a shared object, can appear unordered in the event log. This happens because the access to that object is unsynchronised, as when creating and stealing a spark or blackholing and suspending a thunk. Even if this makes replaying the execution harder, we have at least one guarantee: two events related to the same object can appear unordered, but never by more that one event. That is, if an event A in capability zero that occurs as a response to an event B in capability 1 has a smaller timestamp, it will be always larger than the previous event C in capability 1. This proposition holds because A could only have happened by interacting with the object either created or modified by the event B and the events are always logged after they happen.

PARALLELISM Dealing with parallelism is the most complex part of the replay system because of the interactions that need to be taken into account at replay time. In section 4.2.2, we saw that all possible combinations of the execution interleaving of two or more threads have enough events to distinguish them. The problem at replay time is that events can occur unordered, so that the information may not be available at the right time. To solve this, we check that, when a thread is about to evaluate a thunk or a blackhole, it can decide which path it took in the original execution.

During replay, when entering a thunk, a thread always calls the function `replayEnter()` to check if the thunk should have been previously blackholed or else it can enter it normally.

Listing 14: Pseudocode for replaying a thread entering a thunk

```
1  void replayEnter(Capability *cap, StgClosure *p) {
2      int thread = thunkThread(p);
3      Event *ev = readEvent();
4      // enter blackhole
5      if (// explicit event
6          (ev == BLACKHOLE_WHNF) ||
7           ev == MSG_BLACKHOLE ||
8           ev == BLACKHOLE_THUNK)) &&
9           eventId(ce) == id) ||
10         // no enter event when it should have been one
11         (capThunkId(id) != cap->no &&
12          (ev != ENTER_THUNK || ev->id != id))) {
13         Capability *owner = find owner capability;
14         replaySyncThunk(owner, p);
15         ENTER(p);
16     }
17
18     // enter thunk
19     if (first thread entering p) {
20         saveSpark(p);
21         shared = thread != cap->thread->id;
22         if (shared) {
23              new = REPLAY_SHARED_TSO;
```

```
24          } else {
25              new = REPLAY_TSO_ATOM;
26          }
27          ((StgInd *)p)->indirectee = REPLAY_SET_TSO(id, new, cap
                ->thread);
28          write_barrier();
29          SET_INFO(p, &__stg_EAGER_BLACKHOLE_info);
30      }
31
32      // explicit event
33      if (ev == EVENT_ENTER_THUNK && ev->id == id) {
34          traceEvent(ENTER_THUNK, id, p);
35      }
36  }
```

This function (see listing 14) reenters the thunk (line 15) after synchronising with the thread in charge of blackholing the thunk if necessary, or continues execution. The first case can be detected with just two checks: if the next event is related to blackholing and its id matches the one of the current thunk (lines 5–9) or if another thread allocated the thunk and there is no enter/blackhole event related to this thunk (10–12). This last check is effective because, if the thunk was entered, the capability id encoded in the thunk id would not match the current capability and an event would have been emitted. In both cases, the capability to synchronise with can be obtained either from an *enter thunk* event emitted in one of the other capabilities or using `capThunkId()` (see listing 10) if it was entered in the same capability it was allocated (so that no event was emitted in that case).

When none of the previous conditions are met, it means that the thread entered the thunk. In this case, the first thread doing this will make a copy of the thunk, blackhole it and trace the event if needed (lines 19–35). If other threads enter the same thunk concurrently, they will just proceed with the event emission (lines 32–35). Checking for an explicit *enter thunk* event in the event log covers the case where we reverse the execution during replay, so that the second thread enters the thunk first, and because it may have been allocated by that thread, it does not emit an event.

If, instead of a thunk, a thread tries to evaluate a blackhole, we need to check whether a duplicated evaluation occurred (which cannot happen in replay because execution is serialised), or whether it was a real blackhole in the original execution. `replayBlackHole()` in listing 15 works similarly to `replayEnter()` in listing 14, forcing a synchronisation and reevaluation when needed or continuing the execution otherwise.

Listing 15: Pseudocode for replaying a thread entering a blackhole

```
1  void replayBlackHole(Capability *cap, StgClosure *bh) {
2     ev = readEvent();
3     id = thunkId(bh);
4
5      // enter thunk
6      if (ev == EVENT_ENTER_THUNK && ev->id == id) {
7          p = restoreSpark(cap, bh);
8
9      // enter blackhole
10     } else if (// explicit event
11                 (ev == BLACKHOLE_WHNF) ||
12                  ev == MSG_BLACKHOLE ||
13                  ev == BLACKHOLE_THUNK) &&
14                  eventId(ce) == id) ||
15                 // untagged BH
16                 thunkAtom(bh) == 0) {
17         if (((ev == BLACKHOLE_WHNF ||
18               ev == BLACKHOLE_THUNK) &&
19               thunkAtom(bh) != REPLAY_PTR_ATOM) ||
20             (thunkAtom(bh) == 0 &&
21              GET_CLOSURE_TAG(bh->indirectee) == 0)) {
22             Capability *owner = findBHOwner(cap, bh);
23             replaySyncThunk(owner, bh);
24         }
25         p = bh;
26
27     // no event
28     } else if (thunkAtom(bh) == REPLAY_SHARED_TSO ||
29                 thunkAtom(bh) == REPLAY_PTR) {
30         // there would be an event otherwise
```

```
31          p = restoreSpark(cap, bh);
32      }
33
34      ENTER(p);
35  }
```

First, we check if the next event is related to the current thunk (lines 5–6 and lines 9–14). If it is a *enter thunk* event (line 6), we must restore the blackhole to its original thunk (line 7) using the contents previously saved in `replayEnter()` (see listing 14). Alternatively, if we find a blackhole-related event (lines 9–14), we need to check if the blackhole is not yet pointing to the expected value: a WHNF result (lines 17 and 19) or another thunk (lines 18–19). If we expect a thread or blocking queue, no further checks are needed: the thunk was previously blackholed, and it cannot be updated because the thunk's owner is the one responsible for enqueuing the threads blocking on the it, which we would be about to do. When the corresponding result value is not available yet, we synchronise with the blackhole owner (lines 24–25) similarly as we do when entering a thunk. The call to `findBHOwner()` searches for the owner capability by inspecting the thread id if the thunk has not been updated at all, or the thread or blocking queue if it has been blackholed.

If we do not find an event related to the blackhole there are two possibilities. First, when the blackhole is not tagged with a replay atom (lines 15–16), the blackhole must correspond to an evaluated thunk, so that we only synchronise if it has not been updated yet (line 21). Second, if the tag would force the emission of an event (lines 28–29), it means that the thunk was entered and it is also restored (line 31).

Replaying collision events is much easier because they occur unconditionally. When a thunk is going to be updated, if a collision event is expected, the thunk's result is checked in case we need to synchronise with other thread, in a similar way as described earlier.

Apart from thunk-related events, we also need to be able to consistently replay all spark-related events. Besides *create spark* which is emitted at any point when `par` is used, the other events are occur either when pruning the spark queue or when a spark thread searches for sparks to run (section 3.1.2). We described how the spark pool is implemented

using a dequeue with lock-free access to reduce the concurrency overhead to the minimum in section 3.1.2. Because of this we cannot rely on locking to ensure an ordered access when replaying, as we did with messages and capability ownership. Instead, we make use of the thunk identifiers to locate the needed sparks.

A spark thread looks for sparks using `findSpark()`. For our replayer, we reimplemented the whole function as shown in listing 16:

Listing 16: Pseudocode for spark stealing

```
1  StgClosure *replayFindSpark(Capability *cap) {
2      Capability *owner;
3      StgClosure *spark;
4      int id;
5      Event *ev = readEvent();
6
7      while (ce == SPARK_FIZZLE)) {
8          id = eventId(ev);
9          owner = capThunkId(id);
10         prepareSpark(owner, id);
11         spark = tryStealSpark(owner->sparks);
12         ...
13         replayStealSpark(cap, spark);
14         traceEvent(SPARK_FIZZLE, spark);
15         ev = readEvent();
16     }
17
18     if (ce == SPARK_RUN) || ce == SPARK_STEAL) {
19         id = ev->id;
20         owner = capThunkId(id);
21         prepareSpark(owner, id);
22         spark = tryStealSpark(owner->sparks);
23         replayStealSpark(cap, spark);
24         traceEvent(ev->tag, id);
25         return spark;
26     }
27     return NULL;
28 }
```

As can be seen, we rely on knowing the thunk id to steal the correct spark (line 8 and 19) and on the encoding of the source capability in the thunk id (line 9 and 20). Because sparks may not be yet available, there may be some synchronisation to be done between capabilities, which is handled in `prepareSpark()`:

Listing 17: Pseudocode for readying a spark in its spark pool

```
1 prepareSpark(Capability *cap, int id) {
2     while (id not found in cap->sparks) {
3         replaySync(cap, myTask());
4     }
5     if (topSpark(cap->sparks) != id) {
6         find id and move to top;
7     }
8 }
```

Because of the unsynchronised access to the spark pool described above, the events that create a spark and steal that same spark from another capability may be emitted unordered (stealing the spark before being created). We need to check this case (line 2) and also the case where different *steal spark* events are emitted in the wrong order so that, when replaying them we would find that the spark to be stolen is not at the top of the pool (line 5). The latter is solved just by moving the right spark (we know its id) to the top. If the spark is not found in its capability, we need to synchronise with the spark owner (line 3).

4.3 SUMMARY

In this chapter we described the implementation of an execution replay system, integrated in the RTS of GHC. We analysed the events responsible for nondeterministic behaviour, classified into initial setup, concurrency, thread scheduling, parallelism and GC events. For all of them, we described what information they needed to provided and why it was needed for replay. A summary table was presented in table 2.

The relatively low amount of events was due to the fact that our replay mechanism is designed to serve as the foundation of new profiling tools,

so that we are only interested in replaying the parallel behaviour of the execution. While this meant that the amount of information collected is small, it also implied a challenge to correctly replay the execution. In particular, it meant that events from different capabilities can appear unordered, or some information is not recorded at all, as in the case of thunk updates. We solved this problem by adding the ability to pause the replay of an event and advance the replay of another task to get it in sync or, in a few cases, intelligently placing the emission of some events so that their ordering follows the one used to coordinate access to protected data structures in GHC.

We claimed that the use of a purely functional language, and having profiling as our use case would allow a low overhead recording phase. In the next chapter, we validate our design with a performance analysis and also present two examples that showcase how the system can be used to build profiling tools.

5 EVALUATION

In this chapter we proceed to evaluate our profiling technique in two ways. First, we analyse the runtime overhead of the recording phase in a number of benchmarks in section 5.1. Then, we analyse some Parallel Haskell programs by building on our profiling infrastructure in section 5.2, to showcase how more advanced and user friendly profiling tools can be build on top of our work.

5.1 PERFORMANCE EVALUATION

In order to evaluate the performance impact of the replay phase of our execution replay (ER) implementation, we have selected a number of programs frequently used for benchmarking in Parallel Haskell research [15, 76]. These are programs that belong to the parallel directory of the *nofib* benchmark suite[1] [111]. We have made a few modifications to increase the execution time of some of them over 3s. All of these changes are available at `https://github.com/hferreiro/nofib/tree/replay`.

All tests were run in a desktop PC with an Intel Core 2 Quad CPU clocked at 2.40GHz with 32KB and 4MB level 1 and 2 caches, respectively, and 4GB of main memory. The benchmarks were run in Debian GNU/Linux 8.0 (Jessie), compiled for x86-64 architecture. For each test, the result is the mean elapsed time of 5 runs, as computed by the `nofib-analyse` program found in the mentioned benchmark suite.

We have measured the performance impact of the recording phase of ER and summarised the results in table 3. The baseline is the program compiled using the 7.8.4 release of GHC with the same RTS configuration (i.e. the threaded and event logging runtime enabled) and the GHC libraries compiled for eager blackholing. In each column, Δ Time measures

1 `http://git.haskell.org/nofib.git`.

the percentual difference in elapsed time of the benchmark compiled with a ER-enabled GHC[2] in comparison with the baseline. The ER version uses the RTS flags `+lsf`. Both versions use the appropriate `-Nn` RTS flags so that each column compares the execution in a different number of cores.

Table 3: Performance impact of ER event logging

benchmark	Δ Time `-N1` (%)	Δ Time `-N2` (%)	Δ Time `-N4` (%)
blackscoles	+08.6	+09.2	+09.1
coins	+13.1	+12.3	+21.0
mandel	+24.2	+23.4	+24.5
matmult	-03.1	+02.9	-01.4
minimax	+54.9	+39.9	+37.9
nbody	+08.1	+08.8	+10.6
parfib	+60.5	+119.1	+142.6
partree	+47.7	+34.0	+29.5
prsa	+03.2	-14.6	-00.3
queens	+03.8	+03.8	+15.4
sumeuler	+03.2	+03.2	+03.4

If we omit, the results from parfib, ER-enabled executions have an overhead ranging from slightly faster than the baseline execution up to around 40% for parallel executions and around 55% running in a single core. This results are good if compared with other state-of-the-art execution replay systems. Also, despite of the individual results, the overhead does not change a lot when increasing the number of cores.

The benchmark with the worse results if parfib. This program runs the Fibonacci function using a threshold to limit its parallelism in a similar way as listing 1. If we analyse its execution, we find that the size of the generated event log is over 350MiB. This is mainly due to the large

2 https://github.com/hferreiro/replay.

number of sparks created: over 5,000,000. If we reduce that number by increasing the threshold value, the overhead of the ER version is negligible (below 3%). The reason for having such a low threshold is that the performance of the parallel execution gives better speedups when reducing the threshold to such a low value. That amount parallelism is not actually needed for such a simple program and the details of why this happens are analysed in section 5.2.1.

Suspecting that much of the performance overhead of the ER recording lies in the implementation of thunk updates, and not just the amount of logging, we collected in table 4 a comparison of the execution of a specially prepared ER-enabled GHC version where the only changes are the ones introduced by thunk tagging. That is, thunks ids are created and stored in the thunk's result when allocating them and blackholing a thunk overwrites the thunk's result with the thread id performing the update.

Table 4: Performance impact of thunk tagging

benchmark	Δ Time `-N1` (%)	Δ Time `-N2` (%)	Δ Time `-N4` (%)
blackscoles	+05.5	+05.0	+04.9
coins	+06.9	+09.0	+04.2
mandel	+12.1	+11.9	+09.4
matmult	+00.5	+01.5	-00.6
minimax	+27.2	+26.1	+24.4
nbody	+00.0	+00.2	+00.6
parfib	-03.4	-03.6	-05.1
partree	+29.6	+20.1	+15.4
prsa	-00.4	-17.8	-04.7
queens	+04.8	+04.7	+02.0
sumeuler	+02.8	+02.7	+02.5

As can be seen, in most cases, half of the performance overhead is due to the code changes introduced to record thunk updates. Additionally, when can see in table 4 that, in the case of parfib, all the overhead was due to the recording of the events.

In conclusion, we compare favourably with current state of the art execution replay systems that generally incur in 10x to 100x slowdowns when recording multithreaded applications.

5.2 USE EVALUATION

In this section, we showcase ways in which some of runtime-related information that we identified as relevant for a profiling tool in chapter 4 can be gathered. In order to do that, we selected two simple problems with some performance problems and whose behaviour is difficult to explain. We focus on the analysis of execution time and its relation with source code expressions. This information is used in section 5.2.1 to discover useless work in the form of *duplicated thunk evaluation* and in section 5.2.2 to analyse the cause of its *limited parallelism* and to develop an improved version.

5.2.1 *Finding duplicate work*

In chapter 3 we described how a lazy functional language can avoid duplicated evaluation of shared thunks by using the mechanism known as blackholing. Two versions of this technique are available in GHC: lazy and eager blackholing. When running parallel programs, the GHC developers recommend compiling with the `-feager-blackholing` command line option to activate the second version. In this mode, almost all possible duplication is avoided, although there is still a small window of opportunity for two threads entering the same thunk at the same time. Lazy blackholing is used as a default to avoid any performance overhead in sequential programs or concurrent programs not used for parallel computation. In this section we will analyse the impact that duplicate

evaluation can have in an otherwise simple parallel program by using ER.

The following is the code listing for a parallel implementation of the Fibonacci sequence found in the `parfib` nofib benchmark suite:

Listing 18: Code for the `parfib` benchmark from nofib

```
1  import System.Environment (getArgs)
2  import Control.Parallel
3
4  main = do [arg1,arg2] <- getArgs
5            let
6              n = read arg1 :: Int  -- input for nfib
7              t = read arg2 :: Int  -- threshold
8              res = parfib n t
9            putStrLn ("parfib " ++ show n ++ " = " ++ show res)
10
11 -- parallel version of the code with a threshold
12 parfib :: Int -> Int -> Int
13 parfib n t | n <= t = nfib n
14            | otherwise = n1 `par` (n2 `pseq` n1 + n2 + 1)
15                        where n1 = parfib (n-1) t
16                              n2 = parfib (n-2) t
17
18 -- sequential version of the code
19 nfib :: Int -> Int
20 nfib 0 = 1
21 nfib 1 = 1
22 nfib x = nfib (x-2) + nfib (x-1) + 1
```

This program contains two versions of the Fibonacci function. The function `nfib` is a sequential recursive function that returns the `n`th number of the sequence. `parfib` consists in the same structure of `nfib`, but creates a parallel task in each iteration of the algorithm up to the limit `t` (line 14). Then, it calls the sequential version (line 13). The `main` function reads the requested Fibonacci number and the selected threshold as command line inputs.

This program is a perfect example to illustrate parallel programming in Haskell because of its simple divide and conquer structure. It would

be reasonable to expect linear speedups with the number of cores, given that the work is split almost perfectly in each iteration, and the parallel threads can start stealing available work since the first iteration.

We run the benchmark using the nofib infrastructure. The options used where `EXTRA_HC_OPTS="-O2 -threaded"` to use the threaded runtime and `PROG_ARGS="43 25"` as the program arguments. In table 5 we summarise the results giving the obtained speedups when using a different number of cores. To obtain results for different cores, we used the option `EXTRA_RUNTEST_OPTS="+RTS -Nc -RTS"` where `c` was substituted by the number of cores to use.

Table 5: Parallel performance of `parfib`

No. cores	Speedup
2	1.617
4	3.519

Even though the results are very good, they do not match our expectations. ThreadScope provides us with the following graph for the runtime behaviour of the program:

The results of this tool indicate that all the computer cores are used to its fullest, with no garbage collection happening at all. There are no hints to what the cause for the underperformance. What is worse, the output of ThreadScope would suggest that the speedup is linear.

To investigate the effect of duplicate computations on this program we used our ER framework. As described in chapter 4, lazy blackholing can only be replayed under certain conditions. The Fibonacci program is one of those in which all shared expressions are only created by sparking because their results are evaluated to WHNF values and there is no risk of unsharing duplicated results.

Our goal is to generate an enhanced event log where all duplicate computations are logged and also provide timestamps of the exact point where thunks are entered and finally updated to a WHNF value. Then,

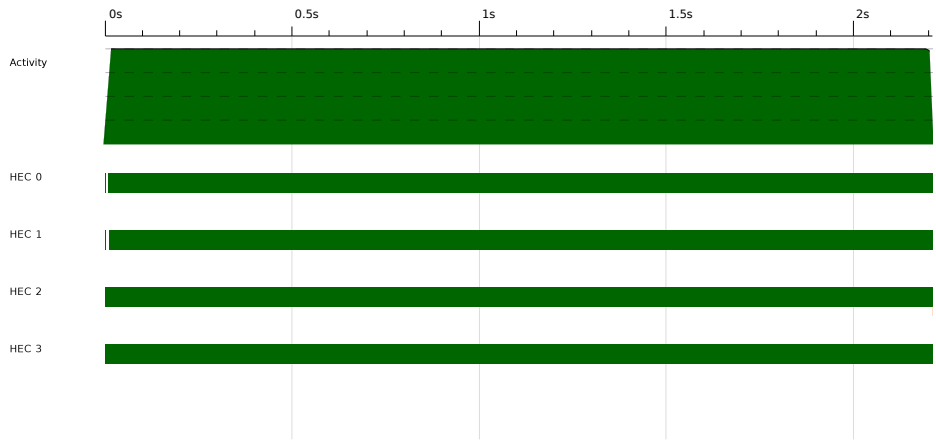


Figure 4: ThreadScope graph of a Fibonacci execution

an ad-hoc program is used to calculate what thunks were needlessly evaluated and the amount of time used on those evaluations.

The first part required the definition of several new events in our tool. Part of the thunks that are evaluated repeatedly is covered with *suspend computation* and the two collision events. As described in section 4.2.2, the first event is emitted in the case that the duplicated evaluation is detected at the time the thread is paused. In case the thread paused and the thunk was updated later, it will emit a *collision event* when trying to update it.

For short computations in which the thread did not stopped, the update overwrites whatever is in the thunk's result space. To record this case, we emit a new *duplicated thunk* event during replay when the the thunk is updated and after inspecting the thunk's result for an already stored result.

The lifetime of sparks is tracked with the existing event *enter thunk* and a new *thunk updated* event emitted at the time a spark is updated to WHNF. The logging of these new events was implemented under a new flag to GHC's RTS. Running a program with `+RTS --dup` will enable ER and emit the new events.

With this new events we have all the information needed to be able to find out the amount of duplication in a particular execution. For this purpose, we developed a tool that calculates the time wasted in duplicated and cancelled thunks per capability. As an additional functionality, we use the tool to generate an especially purposed event log that can be fed to `ghc-events-analyze`[3] to visualise duplicated computations on top of a similar graph as that of ThreadScope.

The wasted time computation works in two phases. After reading the replay log with the added duplicate events it calculates the extended set of duplicated sparks. That is the set of the originally duplicated and cancelled sparks and the sparks that were created while performing the duplicated evaluation and can therefore be considered as wasted work too. Then, it merges the original event log with the replayed log to assign correct timestamps to the *thunk updated* events and calculates the time it took to evaluate the previously calculated set of duplicate sparks.

For the first phase, the list of wasted computations consists of a dictionary that maps the thunk id to the thread ids of the threads updating duplicated thunks. We did not considered any specific algorithm looking for performance and simply used a multiple-pass process. Initially, we start with the list of thunks explicitly tracked as either duplicated, suspended or discarded with a collision. This list is constructed by gathering the information directly from those events. The *suspended computation* and collision events will only happen for the threads that cancel the evaluation of the thunk and the information does not need to be refined further. In the case of duplicated sparks, we have an event for each duplicate update but we need to identify which thunk was entered first to keep as the *real* update. In order to do that, we traverse the sorted event log and, after seeing each *enter thunk* event once, we save the rest of the updates to the same spark as duplicates with the thread performing the evaluation.

3 Tool similar to ThreadScope available at `https://hackage.haskell.org/package/ghc-events-analyze`.

Taking *dups* as the list constructed as described and *used* as the mapping from thunk id to threads entering it, we build the extended set of duplicated sparks *extended* as follows:

new = dups
while *new* is not empty
  extended += new
  clear *new*
  for each *event* in the event log
    if *event* is *run thread(t)* and *t* is in *new*
      thread = t and *spark = sparks[t]*
    if *event* is *enter thunk(id)* and *spark* is not set and *t* in *new[id]*
      spark = id and *sparks[t] = id*
    if *event* is *create spark(*id*)* and *spark* is set
      new[id] = used[id] - thread

In this way, *extended* saves every spark created by a duplicate spark and the threads that update it other that the current thread. This list will contain many superfluous sparks that, while created in this way, they are also updated within the same thread that created them, so that their evaluation will be part of the evaluation of an enclosing duplicate thunk. We clean *extended* by performing a similar traversal as described before where entering a spark while evaluating a duplicate spark will remove the entered spark and its thread from *extended*.

Having built *extended*, calculating the amount of wasted time is quite easy. Both event logs are merged so that the new *thunk updated* event is assigned the timestamp of the event before and after it. A single traversal over the event log will save, for every duplicated thunk, the timestamp of *enter thunk* event and accumulate the spark lifetime when encountering the corresponding *thunk updated*, *suspend computation* or collision events.

For a Fibonacci execution with the following stats:

```
$ ./fib 43 25 +RTS -s -N4
...
INIT    time    0.001s  (  0.001s elapsed)
MUT     time    8.714s  (  2.210s elapsed)
```

```
GC      time    0.000s  (  0.000s elapsed)
EXIT    time    0.071s  (  0.024s elapsed)
Total   time    8.788s  (  2.235s elapsed)
...
```

we obtain the following output:

```
cap 0: 1092-1092ms
cap 1: 332-335ms
cap 2: 381-381ms
cap 3: 11-11ms

Mean: 454-455ms
```

The mean time is the maximum amount of time that we could gain if the duplicated computations could be removed evenly in all capabilities. It can be observed that the accuracy of the result is high, with a timestamp range of 1ms.

We mentioned that our tool is also capable of generating a new event log that can be used with `ghc-events-analyse` to visualise the calculated data. A graph built like that, showing the program execution and where duplication happens is shown in figure 5.

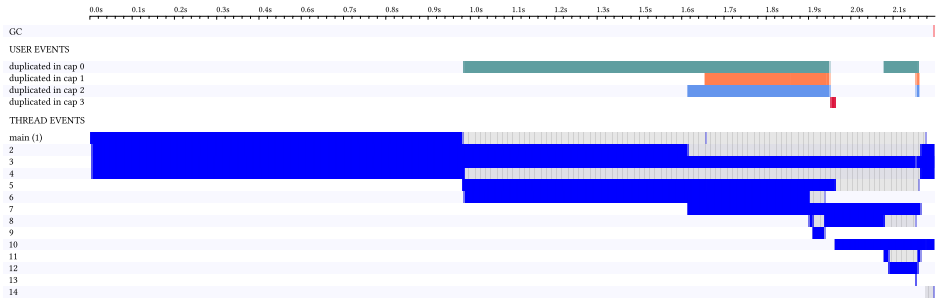


Figure 5: Visualisation of duplicated thunk in a Fibonacci execution

It shows very similar information to that of ThreadScope. On the horizontal axis indicates the timeline of the program execution while the vertical axis classifies information per thread. Each thread timeline has an opacity proportional to the thread runtime in a given timeslice. In this

case, as also shown in ThreadScope, all colours are completely opaque for most of their execution, showing no idle or GC time in between.

There are two areas in the vertical axis. First, under the label *USER EVENTS*, the timeline of user-defined events created in the program execution is shown. We used this functionality while generating the new event log to mark what amount of time was spent evaluating duplicated thunks. The main informations, under the label *THREAD EVENTS*, corresponds to the normal timeline of every thread in the program. We need to remember that a single spark thread is used to run multiple sparks and that is why there are so little threads in comparison with thousands of sparks created. On the other hand, because the user events correspond to the evaluation of single thunks, we know that their timeline maps to thunks in a one-to-one fashion.

In a quick glance, we can observe that, even though there are always four threads completely busy (blue colour); during half of the program, much of the work done is useless (green, orange and light blue), as it will be eventually discarded when each duplicated thunk evaluation is discovered.

We now *know* that the reason for the slowdown in the Fibonacci code is related to the way that lazy blackholing allows some duplicate evaluation to happen. It may result surprising that such a big amount of wasted work is performed. It needs to be said that for this behaviour to happen some conditions need to be met at the same time:

1. Many sparks have to be created in very little time so that, even if the duplicated computations are detected and cancelled, many sparks were created from those duplicates before suspension.

2. The program cannot allocate too fast, so that the created sparks are not garbage collected too often.

This is exactly the case of Fibonacci. The sequential code is optimised so that the function parameters are unboxed and no allocation is performed. Only the parallel code performs some allocation by creating a thunk to hold the result of a parallel spark evaluation. Even so, that

amount is limited by the configured threshold. This fact results in executions with no garbage collection at all, so that any spark created from thunks that may be eventually suspended will survive until the end of the program.

As said at the beginning of this section we can avoid most duplicate evaluation by enabling eager blackholing when compiling our program. table 6 summarises the parallel speedups of the program compiled with `-feager-blackholing`:

Table 6: Parallel performance of `parfib` using eager blackholing

No. cores	Speedup
2	1.998
4	3.947

As originally expected, the speedups are linear with the number of cores. Even though eager blackholing should be considered as the default for parallel programs, we believe that the ability to properly quantify and verify the exact reasons for the loss of performance in the previous version is an important contribution. In particular, it allows us to explain why the parfib benchmark of nofib was configured to use a threshold lower than what was actually needed, which was the reason of the excessive recording overhead.

At this point, one could wonder why is ER needed and whether the newly added events could be incorporated to the normal events logged as part of the default event logging implementation. There are several reasons supporting the need for ER:

- Detecting duplicate evaluation is not always possible in the presence of concurrency. Event if a thread could check if a particular thunk is updated or not before performing its update, several threads can do the check at the same time without realising that all of them will finally perform the update. The only way to avoid this situation is to either use locking or compare-and-swap prim-

itives. The blackholing design of GHC was developed to avoid the performance overhead of those two options.

- Though it was not needed for this particular program, there needs to be a mechanism to find out whether the extended set of duplicate sparks is valid. This situation would happen when the new sparks created from a thunk that will eventually be cancelled or detected as duplicate are used in the final result. It can happen when the expression sparked is not related to the duplicated/cancelled expression, such as a CAF or a binding defined in the same or higher level. This is easily done with ER by checking whether they are entered by non-duplicated thunks after being updated to WHNF.

- The cost to emit the new events and keep the needed information around is high. A check would need to be placed in the thunk enter code that would test every thunk for its shared and evaluation state. Also, a mapping from thread pointer to id, as used in the replayer, would need to be kept between each GC.

5.2.2 *Quicksort*

In order to show how ER can be used for performance debugging of non-trivial parallel programs, we use the simple Quicksort implementation presented in listing 19. Quicksort 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.

Listing 19: Simple parallel Quicksort implementation

```
1 psort :: Int -> [Int] -> [Int]
2 psort _ []     = []
3 psort n (x:xs)
4   | n > 0      = hi' `par` lo' `pseq` (lo' ++ x : hi')
5   | otherwise  = seqSort (x:xs)
6   where
7     (lo, hi) = partition (<x) xs
```

```
8       lo' = psort (n-1) lo
9       hi' = force (psort (n-1) hi)
```

The rationale behind this attempt at parallelising 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` to make sure that each parallel thread completely evaluates its sublist. In addition, by using the `n` parameter, we control the amount of parallelism generated, so that after a certain point is reached in the recursion depth, 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.

Because we are analysing the parallel implementation we chose to use an already balanced input list, and also force the evaluation of the list before measuring the execution time of the sorting process. However, no matter what value for `n` we choose, the speedups of `psort` that we obtain are very poor, not even achieving a speedup of `2` in up to 8 cores.

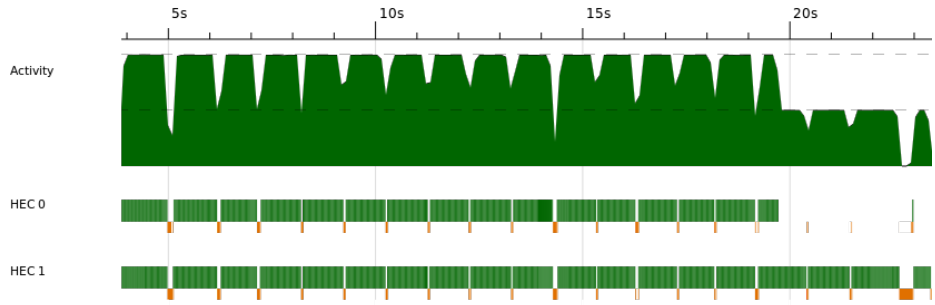


Figure 6: ThreadScope profile of `psort`

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 6 shows an execution profile of the program. From this profile we can observe that the program behaves reasonably well most of the execution. Then, towards the end, there is some serialisation where only one thread at a time is doing evaluation. However, ThreadScope does not pro-

vide us any hints about where do these problems come from, e.g. what part of the program is responsible for the final sequential phase. Based on the knowledge 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.

In order to come up with a better parallel program, we first made some optimisations to the sequential version, show in listing 20. 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.

Listing 20: Optimised sequential Quicksort

```
1  qsort1 :: [Int] -> [Int]
2  qsort1 xs = seqSort xs []
3    where seqSort []     zs = zs
4          seqSort (x:xs) zs = seqSort lo (x : seqSort hi zs)
5            where (lo,hi) = partition x xs
6
7  partition :: Int -> [Int] -> ([Int],[Int])
8  partition x xs = go xs [] []
9    where go []     ts fs = (ts,fs)
10         go (y:ys) ts fs
11           | y < x       = go ys (y:ts) fs
12           | otherwise   = go ys ts     (y:fs)
```

Similarly to the first time, we tried to naively parallelise this code in the same way (listing 21). Given the changes mentioned above, we expected to avoid at least the sequential phase that occurs at the end of the execution.

Listing 21: Parallel Quicksort using an accumulator

```
psort1 :: Int -> [Int] -> [Int]
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 had 8MB of L2 cache, that was shared between all of its cores. The total amount of RAM was 64GB. In table 7, we show the speedups obtained by taking the mean time over five runs of each program with the same input, a list consisting of 10 million elements.

Table 7: Parallel performance of `psort1`

No. cores	Speedup
2	1.49
4	0.78
8	0.60

In this case, we can observe that we are actually getting significant slowdowns as we use more cores.

In order to debug the performance of this program, we used again ThreadScope to get an overview of the thread activity. figure 7 shows the profile from ThreadScope after running our program using two cores.

We can see that we still have the same serialisation problem that we had in the initial parallel version in listing 19, and that getting rid of the ++ operator did not help at all. Additionally, there is a pause in

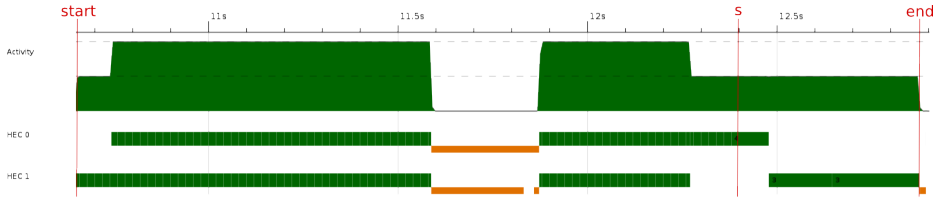


Figure 7: ThreadScope profile of `psort1`

the execution corresponding to a major garbage collection phase. The big amount of input data, coupled with the fact that we set up a large allocation area, is responsible for this behaviour. `psort` does not present this gap because, due to its inefficient sequential implementation, we had to provide a much smaller input list.

We now used ER to discover which part of the program is responsible for the sequential phase at the end of the execution. We introduce a language primitive that allows to tag expressions at the source code level to be able to log timestamps when the evaluation of an annotated expression is finished and to analyse the output produced. By using ER, we were sure that the same execution was reproduced and so that 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 7}). We then replayed the program and processed its output to obtain the following report:

Listing 22: Execution report from `psort1`

```
1    188.020 (   93.914) cap 0: partition [10.837.539]
2  1.240.278 (1.052.258) cap 0:   seqSort [11.889.797]
3  1.747.763 (  507.485) cap 0:   seqSort [12.397.282]
4  1.747.766 (        3) cap 0:     force [12.397.285]
5  1.828.627 (   80.861) cap 0:     force [12.478.146]
6
7          0 (        0) cap 1:     start [10.649.519]
8     94.106 (   94.106) cap 1: partition [10.743.625]
9    170.970 (   76.864) cap 1: partition [10.820.489]
10   732.638 (  561.668) cap 1:   seqSort [11.382.157]
```

```
11    1.621.573 (  888.935) cap 1:   seqSort [12.271.092]
12    1.996.394 (  374.821) cap 1:     force [12.645.913]
13    2.225.849 (  229.455) cap 1:     force [12.875.368]
14    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 relative 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 zero is 12.397s (checkpoint `s` in figure 7) 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 that is passed in the accumulator `zs`. This sublist is sorted by another thread, so the thread evaluating the `force` call becomes 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 could be the reason why the speedups get worse as we add more cores.

In the end, the same behaviour that 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 data type `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 the list as fully evaluated. Together with this new type, we introduced a `toList :: List a -> [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:

Listing 23: `toList` function

```
data List a  =  Nil |  Cons a (List a) |  Done [a]

toList :: List a -> [a]
toList Nil          = []
toList (Cons x xs)  =  let xs' = toList xs
                       in x `seq` xs' `seq` x:xs'
toList (Done xs)    = xs
```

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:

Listing 24: Improved parallel Quicksort implementation

```
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 table 8. We can observe much better speedups than for `psort1`. For two cores, the speedup is almost

linear. When we add more cores, speedup is further increased, but the relative performance is decreased. This 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.

Table 8: Speedups of the different parallel versions of Quicksort

No. cores	`psort`	`psort1`	`psort2`
2	1.69	1.49	1.90
4	1.71	0.78	2.35
8	1.51	0.60	2.75

This analysis is only a superficial one of some function execution times and how they are related to each other. By using ER we would be able to continue the analysis of the execution behaviour of this program with a more in-depth study of the reasons of some of the behaviour present in the application that we did not cover. ER would be a guarantee that any further information obtained comes from executions that behave in the same way as the one from which we obtained the report in listing 22.

6

CONCLUSIONS AND FUTURE WORK

The main objective of this thesis was to improve the state of the art in profiling tools for parallel execution of non-strict purely functional programs. We identified a common characteristic of previously available profiling tools: either they impose an overhead that makes them unusable on a parallel context, or they provide less useful about the runtime execution than needed for an effective profiling. The thesis of this dissertation was that an implementation of execution replay, tightly integrated in the runtime system of the compiler, could be used as the basis for the development of less intrusive profilers that would allow to gather more detailed data without the performance penalty of the more intrusive tools.

We started by observing the limitations of currently available tools and techniques with respect to the kind of information they provide and the kind of tool they are (i.e. the approach to profiling/debugging they use). By analysing the details of parallel profiling, we identified the execution overhead and nondeterminism as two aspects that limited the usefulness of said tools. We chose execution replay as the mechanism that would overcome those limitations and analysed what runtime information was needed for the correct replay of a non-strict purely functional execution in the form of events. We implemented execution replay in the runtime system of one of the most popular compilers for a functional language (GHC and Haskell) . Our implementation outperformed the state of the art in the aforementioned two aspects, and so paves the way for the development of improved profilers.

We list the specific contributions of our work in section 6.1, analyse its limitations in section 6.2 and discuss future work in section 6.3.

6.1 CONTRIBUTIONS

The research contributions of this thesis are the following:

1. We have identified the source of nondeterminism available during the execution of a parallel purely functional language in the form of runtime events in section 4.2.2. For each event, we have discussed how it contributes to a nondeterministic execution and what information is relevant in the event. From all nondeterminism, we have identified shared thunk evaluation as the greater source of nondeterminism and proposed a scheme to reduce the amount of information logged that nevertheless provides all information needed to know how those thunks were evaluated, described in section 4.2.2.

2. We have implemented an execution replay system tailored to the needs of a profiling framework, described in section 4.2. We did so by integrating the execution replay mechanism in the runtime system of the language's compiler, the source of nondeterministic executions. Because the information required for profiling does not require the level of detail required for debugging concurrent programs, our implementation significantly relaxed the assumptions that we need to make about the replay with regard to those needed when ER is used as a debugging facility. Specifically, we are not restricted to having to reproduce the exact same execution as the original one, but are only interested in the parallel behaviour. This considerably reduces the amount of logging information that is required for replay, and it also enables the use of profiling RTS facilities. For example, the program can be stopped during replay and any runtime analysis or data gathering can be performed without affecting the replay process. The information obtained in this way can later be merged with the original event log to obtain improved profiling data.

3. We have made the first implementation of ER targeting a purely functional language RTS, taking advantage of the language prop-

erties, and analysing its pros and cons, presented in section 4.2. In comparison with conventional ER systems, our implementation does not need to log any memory access beyond shared thunk evaluation. This allows very low overhead in the recording phase, even in the presence of fine-grained parallelism, as is the case of parallel functional programs.

4. We have showcased the use of this new profiling framework to profile the performance of some Parallel Haskell programs, obtaining better data about its runtime behaviour and improving its parallel performance with this knowledge (presented in chapter 5).

6.2 LIMITATIONS

In the description of the recording phase in section 4.2.2 we have discussed some design decisions that limited the applicability of our technique:

- Mixing thunk ids and thread ids in the same space imposes a limit in the amount of both threads and thunks than can be created. This would likely not allow for long-running executions to be replayed in full. An idea to solve this problem would be to use a checkpointing facility. This technique saves the full state of the process in a file and allows to restore the execution from that same point at a later time. In the event of some of the id spaces being nearly full, the replayed could checkpoint the program execution at the beginning of the next GC, and use the GC process to compact the ids still in use.

- Despite advising the usage of eager blackholing to avoid concurrent thunk updates to pass undetected, there is still a very small chance for a thunk being updated by more than one thread without reporting a collision. Because we are replaying purely functional programs, both results should be equivalent (the same WHNF value evaluated to the same depth), so that the result would be correct

even if the update order was reversed. What can make the execution unreplayable is that applying the updates in a different order, or at different times, may make other threads read an intermediate state of the thunk. A technique called probabilistic replay is used by other approaches to perform successful replays even in the presence of partial trace information [44, 45]. The idea is to explore the space of possible executions that fit the recorded log until the program is replayed in full. While this process may take time, it is only a one-time execution. After the first complete replay, the event log can be modified to include the missing information.

- Execution replay allows to perform any kind of analysis or data gathering at runtime but the information has restrictions on how it can be used. In particular, if we are to report timelines with new events emitted during replay, we have to be careful to only trust the timestamps of the original event log and, for the new events, use timestamp ranges as done in section 5.2.1. In relation to this, there may be executions in which the granularity of events do not allow to give very detailed information, e.g. if two consecutive events happened too far apart from each other. This kind of problem could be solved by mapping the replayed execution between those two events into the original event log and interpolating the timestamp of the new events using its relative position.

- Some parallel algorithms may present higher than normal overhead in the logging phase if they use a very large amount of sparking. This was the case of `parfib` in table 3. Even though in that case it was due to choosing the wrong compiler options that forced the programmer to lower the sparking threshold, we do not discard the possibility of programs that require such an amount of parallel tasks to obtain good speedups. For cases like that, we should explore the possibility of removing some spark-related events as is discussed in section 6.3.

6.3 FUTURE WORK

The main line of research that remains open is that of formal models that allow to verify the completeness of the event logging and replay phase. Despite having build a working execution replay system, formal verification of the design would increase confidence on its correctness. The usage of model checking techniques on a formal specification of the record and replay process could be used to verify the correctness of the approach.

Additionally, a formal description of the system may help in defining causality relations between events and reduce the need for some of them. As an example, when a spark is stolen, because we encoded the capability where the spark resides in its id, the event to create the spark may not be needed for replay. We have not explored these possibilities to further reduce the overhead of the recording phase by emitting less events.

Further validation of our approach to profiling would be the development of profiling tools that take advantage of our framework. In particular, one of the claims of this work is that most of the sequential profiling tools analysed in chapter 3 can be adapted to work in a parallel setting by building on top of our work. For example, one commonly used profiler still missing for Parallel Haskell is a memory profiler. The GHC profiling subsystem makes use of cost-centre stacks (see section 3.2.1) and can only work with one processor. Its overheads are over twice the sequential performance which makes it unusable in a parallel program. An implementation that used ER could completely isolate the profiler by storing the extra label work that is attached to every thunk in a separate area as to avoid interference in the profiled executions. Then, during replay, the normal memory profiling process would proceed with the new label locations. The suggested implementation would have the benefit of working with parallel programs and ensuring minimal impact in their execution.

Improvements to our replayer implementation can be made in two areas. First, to add support for concurrency. By adding specific support to the most popular concurrency primitives and libraries, our system

could be extended to support concurrent and distributed applications. Second, to implement parallel replay. The current implementation replays executions single threadedly by following the event log timestamp ordering and coordinating threads through a single *replay scheduler*. The replay performance could be improved by making each program thread be its own scheduler so that they would read events by themselves and only perform coordination tasks when events relate them with other threads.

Last but not least, another research area would be to use ER in combination with the GC research started in Ferreiro et al. [112]. That work explores the online resizing of the GC's nursery to adapt to the dynamic behaviour of applications with respect to its memory requirements, and improve the overall runtime performance. A limitation of that work is that it focuses on sequential execution. Our execution replay technique could be used to analyse the applicability of the techniques developed in the referenced work to parallel applications and develop parallel versions of the nursery resizing algorithms.

A

RESUMO

A.1 OS DESAFÍOS DO PARALELISMO

Desde o comezo da historia da informática, foi investida unha enorme cantidade de investigación no desenvolvemento da tecnoloxía que permite facer cálculos en paralelo como unha forma de mellorar o tempo de execución dos programas. Os límites na escalabilidade dos procesadores de un só núcleo aumentaron recentemente a necesidade de aproveitar o hardware paralelo con máis eficiencia. A duplicación da potencia de cálculo cada dous anos, derivada do aumento de transistores por chip, coñecido como lei de Moore, estancouse hai varios anos cando a miniaturización comezou a encontrar límites físicos como a disipación de calor, o que aumenta o consumo de enerxía e a corrente de fuga [1]. A pesar de que a lei de Moore continúa a ser vixente, o aumento de transistores por chip conséguese agora utilizando arquitecturas multinúcleo. O que antes era cuestión de esperar algún tempo para adquirir un procesador máis rápido, agora require un cambio fundamental na forma en que se escriben os programas para aproveitar a execución concorrente.

Os computadores de escritorio e portátiles de hoxe en día teñen chips con múltiplos núcleos, mais a investigación actual xa está traendo hardware *manycore* e plataformas heteroxéneas, onde a CPU se acompaña con GPUs altamente paralelas [2]. A mellora do paralelismo no hardware desafía os modelos de programación actuais para se manter ao día co aumento da concorrencia. A pesar de que a programación paralela existe desde hai moito tempo, facela fácil de usar e, ao mesmo tempo, proporcionar os mecanismos e ferramentas para entender e mellorar o seu rendemento e escalabilidade, segue a ser un problema de investigación aberto.

O que fai que a programación paralela sexa difícil é que, co fin de obter melloras no tempo de execución con respecto a unha implementación

secuencial, un programa paralelo necesita descompoñer os seus cálculos en tarefas independentes que realicen o seu traballo de forma paralela entre si. Dedúcese entón que o programador necesita as ferramentas e a capacidade para identificar estas tarefas independentes dunha maneira en que a cota de traballo total do programa se distribúa do xeito máis uniformemente posible. Facer isto implica unha gran complexidade, como descubrir dependencias de orde ou localizar tarefas independentes. Ademais, é moi posible que sexa necesario algún tipo de coordinación entre as múltiples tarefas que se executan, debido a que, en xeral, os seus resultados teñen que ser combinados, ou as súas entradas proveñen dunha orixe de datos compartida. Nestes casos, situacións indesexables como os bloqueos mutuos ou *thread starvation* deben ser evitados. Esta complexidade é o motivo polo cal escribir código paralelo non foi totalmente automatizado, ao tempo que explica a existencia de múltiplas bibliotecas e ferramentas creadas para tratar de facilitar a programación e o análise do desempeño dos programas paralelos.

A modo de resumo, co fin de aumentar o rendemento, linguaxes de programación e sistemas de execución deben proporcionar primitivas e/ou bibliotecas paralelas aos programadores para que os programas informáticos poidan aproveitar ao máximo o hardware altamente paralelo dispoñible. Este estilo de programación implica máis retos que a programación secuencial, de modo que as tarefas de depuración e análise destes programas constitúen unha gran parte dos antecitados retos.

A.1.1 *A programación funcional paralela*

Aínda que os modelos de programación paralela permiten estruturar un programa paralelo, a maior parte do traballo é a comprensión do programa e a capacidade de reorganizar e descompoñelo en tarefas máis pequenas con poucas dependencias entre si. Moitas das dificultades para realizar isto están relacionadas coa linguaxe de programación utilizada. Nos paradigmas de programación máis populares, o imperativo e o orientado a obxectos, os *efectos laterais* son unha parte integral do seu modelo de programación. En particular, a mutabilidade dos datos é

unha característica fundamental destes paradigmas que dificulta o razoamento sobre as dependencias de datos e a súa orde, algo fundamental para obter unha estrutura paralela efectiva. Isto é especialmente importante para o futuro inmediato, onde a necesidade de escalar a millares de nodos fai que o problema sexa máis relevante.

Aínda que o paradigma da programación funcional existiu dende que existe a programación imperativa, foi só recentemente que comezou a facerse popular. As linguaxes de programación modernas empregan habitualmente unha serie de características de distintos paradigmas, o que fai difícil delimitar que é o que fai que unha linguaxe sexa considerada funcional ou imperativa. Aínda así, unha linguaxe funcional pódese definir como unha linguaxe de programación en que a principal ferramenta de construción é a definición de funcións, e onde programas complexos se constrúen compoñendo funcións máis simples entre si. A pesar desta definición mínima, cando pensamos nunha linguaxe funcional, habitualmente élle asociada unha serie de abstraccións de alto nivel: funcións de orde superior, recursividade, pureza, transparencia referencial e un sistema de tipos expresivo.

Laziness e Haskell

Existe unha característica das linguaxes funcionais que omitimos até agora: *strictness*. *Strictness* define como se avalía un programa: se a función avalía os seus argumentos antes de avaliar o seu corpo (avaliación *eager* ou estrita) ou se se fai ao contrario (avaliación non estrita), de maneira que os argumentos se avalían baixo demanda, cando son requiridos.

Nesta tese usamos unha linguaxe de programación funcional particular cunha semántica non estrita chamada Haskell [11]. O estándar orixinal de Haskell 98, editado por Peyton Jones and Hughes [12], foi publicado en 1999. Aínda que o informe define Haskell como unha linguaxe funcional non estrita, é moi común usar o termo *lazy* porque a maior parte das súas implementacións usan unha estratexia de avaliación *call-by-need*, na cal se utiliza redución de grafos. Isto significa que non só a avaliación de expresións é posposta, mais tamén que os seus resultados son compartidos nos seus múltiples usos. Debido a que *laziness* é un

detalle de implementación da semántica non estrita e esta convención é xeneralizada, nesta tese usamos os termos de maneira intercambiable, aínda que non sexa enteiramente preciso.

Hai moitos beneficios asociados coa propiedade de *laziness*, entre os cales os máis usuais son os seguintes [13]:

- A capacidade de definir novas estruturas de control. Mentres que outras linguaxes teñen que proporcionalas como parte da definición da linguaxes, unha linguaxe *lazy* permite ao usuario definir as súas estruturas de control como calquera outra función. Por exemplo, Haskell proporciona a seguinte función:

  ```
  when :: (Monad m) => Bool -> m () -> m ()
  ```

  equivalente á rama *then* dunha sentenza condicional. Unha linguaxe que usase unha semántica estrita necesitaría un sistema de macros que fixese substitución textual ou soporte nativo na propia linguaxe, o cal é máis habitual.

- A mellora da modularidade a través da separación de conceptos[1] *Laziness* permite definir produtores de datos de maneira independente dos seus consumidores sen que haxa que preocuparse sobre a eficiencia ou a corrección. De novo, proporcionamos o código dunha función Haskell moi común:

  ```
  any :: (a -> Bool) -> [a] -> Bool
  any p = or . map p
  ```

  Este tipo de reutilización de funcións é imposible nunha linguaxe estrita, que utilizaría operadores da linguaxe con avaliación en curtocircuíto como `||` en C ou `orelse` en Erlang para evitar avaliar toda a lista despois de encontrar o elemento que cumpre a propiedade `p`.

De agora en diante, aínda intentando xeneralizar, referirémonos a unha linguaxe de programación pura e *lazy* cando falemos de programación funcional. En particular, usaremos Haskell e Glasgow parallel Haskell (GpH) [14] como a implementación concreta de Parallel Haskell.

1 Do inglés *separation of concerns*.

Parallel Haskell

GpH introduce dúas primitivas da linguaxe para programar aplicacións paralelas:

```
par :: a -> b -> b

pseq :: a -> b -> b
```

Estas son as primitivas básicas sobre as cales moitos modelos de programación paralela se poden implementar [15]. `par`, tal como é definida polo seu tipo, devolve o seu segundo parámetro. Introduce paralelismo *denotando* que sería útil avaliar o seu primeiro parámetro en paralelo. Os pormenores exactos de como se fai isto son definidos na implementación de Haskell subxacente. Este mecanismo é moi flexible, porque a expresión que posiblemente sexa avaliada en paralelo é tratada como calquera outra expresión en Haskell e, como tal, o seu resultado é compartido por calquera *thread* que o necesite e tamén é reciclado polo colector de lixo. Isto permite paralelizar expresións con diferentes graos de granularidade e usar paralelismo especulativo, confiando no sistema de execución para asegurar que non haxa avaliacións duplicadas e se descarten valores que non sexan usados. É importante decatarse de que a avaliación *lazy* é necesaria para que a avaliación de `par a b` devolva `b` sen forzar a avaliación de `a`. `pseq` complementa a `par` forzando a orde de avaliación de dúas expresións. Utilízase principalmente para asegurarse de que o *thread* que avalía unha expresión usando `par` non avalía inmediatamente a subexpresión que se marca para avaliación paralela.

Para ilustrar como se introduce paralelismo nun programa existente usando estas dúas primitivas, proporcionamos o código dunha implementación paralela de Fibonacci a continuación:

Función de Fibonacci paralela

```
pfib :: Int -> Int
pfib 0 = 1
pfib 1 = 1
pfib n = n1 `par` n2 `pseq` n1 + n2
  where
```

```
    n1 = pfib (n-1)
    n2 = pfib (n-2)
```

Esta versión da función de Fibonacci en paralelo avalía a primeira chamada para calcular o número de Fibonacci anterior, `n1`, en paralelo co segundo, `n2`. Neste caso, `pseq` forza ao *thread* avaliando `pfib` a avaliar `n2` antes de `n1 + n2`. Isto permite que `n1`, que foi *marcado* para ser avaliado en paralelo, sexa avaliado por un *thread* diferente.

A avaliación *lazy* e outras abstraccións da programación funcional proporcionan moitos beneficios á programación paralela e, en particular, ao modelo semiimplícito que proporciona GpH. Seguindo a Hammond and Michelson [10, pp. 1–7], podemos mencionar os seguintes exemplos: facilidade de particionamento, modelo de comunicacións simple, ausencia de bloqueos mutuos, depuración semántico directo, aproveitamento fácil de *pipelining* e outras estruturas de control paralelo.

Dadas estas propiedades, escribir programas paralelos nunha linguaxe funcional pura como GpH é enganosamente fácil. En moitos casos consiste en identificar puntos concretos no código fonte onde computacións de gran groso son avaliadas e tamén indicar ao sistema de execución que as avalíe en paralelo.

O escenario anteriormente descrito parece ideal, especialmente cando se compara con moitos modelos de programación paralelos onde as condicións de carreira[2] e os bloqueos mutuos son situacións habituais das que hai que estar prevido. Por outra parte, as mesmas propiedades que facilitan a programación paralela poden ser un desafío cando se quere mellorar un programa xa paralelo ou resolver problemas de eficiencia. A falta de control de fluxo explícito fai que sexa moi difícil definir unha orde de avaliación cando esta é coñecida. Asemade, o paralelismo implícito pode ser un obstáculo cando se implementan problemas cunha estrutura paralela ben definida. A implementación de `pfib` mostrada anteriormente é un bo exemplo destas limitacións. Un programador relativamente inexperto pode darse conta de que é necesario poñer un límite ás chamadas recursivas que introducen paralelismo en algoritmos de ti-

2 Do inglés *race conditions*.

po *dividir e conquistar*, mais, aínda así, as melloras obtidas na execución paralela non son as esperadas nun exemplo tan simple. Unha explicación deste feito ofrécese na sección 5.2.1.

A.1.2 *Facendo profiling de programas funcionais paralelos*

Habitualmente, nun contexto imperativo, despois de obter unha versión inicial dun programa paralelo, o seguinte paso consiste en usar ferramentas de *profiling* para obter información de execución que axude a comprender o seu comportamento paralelo. Esta información pode incluír tempos de execución das funcións, coordinación de *threads*, contención de *locks* etc. O uso deste tipo de ferramentas pode ser moi útil para que o programador poida identificar e centrarse nas causas máis importantes que afectan ao posible baixo rendemento do programa e tamén entender os problemas que o causan.

O problema cunha linguaxe funcional *lazy* en relación coa programación paralela é a falta de ferramentas para facer *profiling* de maneira útil. Isto ocorre mesmo para programas secuenciais. A causa desta circunstancia é o feito de que intercalar a avaliación de funcións segundo son demandadas impide medir de maneira precisa os seus tempos de execución e fai moi difícil relacionar unha expresión que está sendo avaliada a un punto concreto no tempo. Por exemplo, nunha expresión produtor/-consumidor común como `take 5 primes`, en vez de calcular unha lista de números primos e despois devolver os primeiros 5 valores, a avaliación *lazy* significa que cada un dos 5 primeiros números primos vai ser calculado segundo se consome. Entón, se eses números son imprimidos de maneira separada, `primes` suspendería a súa avaliación entre cada acción de impresión.

Ademais, nun contexto paralelo, calquera técnica que intente resolver este problema vólvese máis difícil de aplicar porque o incremento na cantidade de información por unidade de tempo fai que o sobrecusto de obtela sexa maior. Tamén, o indeterminismo inherente a unha execución paralela significa que o comportamento do programa en execucións consecutivas onde se aplicaron distintos tipos de *profilers* pode variar

substancialmente con respecto á da execución orixinal do programa (sen ningún *profiling*).

Algúns destes problemas non se poden resolver directamente e faise intentando aproximar os seus resultados. Por exemplo, para calcular os tempos de execución de función, unha aproximación estatística pódese obter utilizando centros de custo [17]. Aínda que estes non dan unha medida exacta dos tempos de execución do programa, os centros de custo particionan a execución con porcentaxes de uso de cada expresión e permiten priorizar en que partes do programa hai que centrarse.

Relacionar expresións do código fonte con puntos no tempo e, sobre todo, ordenar as interaccións e o progreso dos *threads*, aínda que máis difícil de obter, resulta máis útil para optimizar un programa para unha execución paralela. Obter esta información é vital para decidir se algunhas tarefas deben ser paralelizadas ou non e definir a súa granularidade. Podemos clasificar este problema como unha instancia da técnica máis xeral de traceamento software. O *traceamento software* consiste en rexistrar, cun sobrecusto baixo, a ocorrencia de eventos que proporcionan información de execución relevante. De novo, neste caso, o modelo de execución dunha linguaxe funcional *lazy* fai que sexa moi complicado comprender unha traza sen procesado do grafo de chamadas dunha execución, onde a avaliación de funcións aparecería mesturada coa avaliación dos seus argumentos. Ademais, proporcionar datos útiles pode requirir inspección custosas do programa en execución e un aumento importante dos eventos traceados, facendo que o seu efecto na execución paralela do programa sexa inasumible.

Até agora, o traballo nesta área foi escaso. Unha extensión do traballo previo de centros de custo [18] foi usada para aplicar a mesma técnica nun contexto paralelo, aínda que, igual que no caso secuencial, require cambios no sistema de execución que teñen un impacto importante sobre o rendemento do programa. Outras técnicas funcionaban facendo unha simulación do programa con un só *thread* e usando código específico para extraer datos de execución de maneira que o programa se puidese optimizar iterativamente antes de despregalo en máquinas paralelas reais [19].

Un dos últimos traballos neste campo involucra un deseño arredor do uso do sistema de traceamento do Glasgow Haskell Compiler para proporcionar unha vista dos eventos de execución no tempo [20]. Aínda que parte desta información é útil, dende o noso punto de vista, é aínda insuficiente para ser considerada unha ferramenta de *profiling* madura e completa.

A.2 UNHA NOVA TÉCNICA DE PROFILING

Na anterior sección identificamos *software tracing* como unha técnica que permite adquirir información de *profiling* para resolver problemas de rendemento nunha execución paralela. Tamén se indicou que a cantidade e tipo de información que se necesita extraer e a natureza indeterminista da execución son un obstáculo nun contorno paralelo.

Tamén identificamos certa información de execución que pode proporcionar datos valiosos para comprender o comportamento de programas paralelos: as interaccións entre *threads* en relación con expresións do código fonte (que expresión é a máis utilizada), traballo perdido (que expresións paralelas foron descartadas) etc. Toda esta información está directamente relacionada co modelo de execución da linguaxe e, debido a iso, é custosa de obter. Cremos que esta información sería útil para facer *profiling* de programas paralelos *lazy*, mais require almacenar unha gran cantidade de datos sobre a execución do programa. As ferramentas actuais non poden facer isto sen afectar a execución do programa dunha maneira que produce que os resultados perdan a súa utilidade con respecto ao seu obxectivo; isto é, a información obtida é válida, pero sería de aplicación a un programa en que o seu comportamento cambiou significativamente en comparación con aquel do que se intentaba mellorar o seu rendemento.

A nosa solución a este problema consiste en aplicar a coñecida técnica de *execution replay* para deseñar unha plataforma de *profiling* que teña unha interferencia mínima na execución e sobre a cal se poidan construír mellores ferramentas para comprender a execución paralela dun programa.

Execution replay (ER) [21, 22] é unha técnica de depuración deseñada para ser usada con programas concorrentes. Debido á súa natureza indeterminista, cada execución dun programa concorrente resulta nunha intercalación diferente de eventos de execución, e posiblemente na execución de distintos camiños do código. ER permite ao programador almacenar unha traza da execución do programa para logo poder reexecutar o mesmo programa paso a paso, seguindo esa traza. A traza do programa encapsula parte do estado do sistema segundo vai cambiando ao longo da execución. Desta maneira, a reexecución do programa pode simular a execución orixinal o máis fielmente posible. Ao facer a reexecución, o programador pode inspeccionar o estado do programa (por exemplo, variables, rexistros, a pila) cos seus valores orixinais en cada paso da execución.

A novidade da nosa aproximación baséase no uso do deseño básico de ER, pero cambia algúns dos seus requisitos para adaptalo ao *profiling* de programas paralelos. No noso deseño, a reexecución do programa é simulada dunha maneira que permite *i*) reproducir as condicións que deron lugar a un desempeño paralelo ineficiente, e *ii*) facer mudanzas na execución do programa para obter información adicional sobre o seu comportamento en execución. Desta maneira, podemos modificar dinamicamente a cantidade e tipo de *profiling* que se fai na reexecución do programa para obter a información requirida sen cambiar o comportamento do programa.

Tendo un mecanismo para obter esta información, permitimos o desenvolvemento de ferramentas de *profiling* máis complexas que as dispoñibles até agora. En particular, debería ser posible estender os custosos *tracers* secuenciais, proporcionar información do *heap* asignando custos por *thread* ou por *core*, ou crear unha biblioteca de novas primitivas da linguaxe de programación que permitan monitorizar o ciclo de vida de expresións do código fonte segundo son avaliadas.

A.3 CONTRIBUCIÓNS

No desenvolvemento desta tese, fixéronse as seguintes contribucións:

1. Analizouse o comportamento en execución dunha linguaxe funcional pura e identificouse a fonte de indeterminismo responsable de comportamento diferente en distintas execucións baixo a forma de eventos en tempo de execución (listados na section 4.2.2 e na table 2).

2. Fíxose unha implementación de ER adaptada ás necesidades dunha plataforma de *profiling*, en vez de ser pensada para facer depuración (descrita na section 4.2).

3. Fíxose a primeira implementation de ER nunha linguaxe funcional pura, aproveitando as propiedades deste tipo de linguaxes e analizando as súas vantaxes e desvantaxes (presentada na section 4.2).

4. Utilizouse con éxito este mecanismo de *profiling* para analizar as características de execución dalgúns programas feitos en Parallel Haskell, obtendo mellor información sobre o seu comportamento e mellorando a súa execución paralela con este coñecemento (presentado no capítulo 5).

BIBLIOGRAPHY

[1] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 24/05/2015) (cit. on pp. 1, 129).

[2] Sparsh Mittal and Jeffrey S. Vetter. 'A Survey of CPU-GPU Heterogeneous Computing Techniques'. In: *ACM Computing Surveys* 47.4 (July 2015), 69:1–69:35 (cit. on pp. 2, 129).

[3] *The OpenMP API specification for parallel programming*. URL: http://openmp.org (visited on 04/09/2015) (cit. on pp. 3, 17).

[4] *MPI: A Message-Passing Interface Standard, Version 3.1*. June 2015. URL: http://www.mpi-forum.org/docs/docs.html (visited on 04/09/2015) (cit. on pp. 4, 17).

[5] *PVM: Parallel Virtual Machine*. URL: http://www.csm.ornl.gov/pvm (visited on 04/09/2015) (cit. on p. 4).

[6] Carl Hewitt, Peter Bishop and Richard Steiger. 'A Universal Modular ACTOR Formalism for Artificial Intelligence'. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245 (cit. on p. 4).

[7] Robin Milner, Joachim Parrow and David Walker. 'A Calculus of Mobile Processes'. In: *Information and Computation* 100.1 (Sept. 1992), pp. 1–77 (cit. on p. 4).

[8] *UPC Language Specifications, Version 1.3*. Nov. 2013. URL: https://upc-lang.org/upc-documentation (visited on 04/09/2015) (cit. on p. 4).

[9] *CUDA: Parallel Programming and Computing Platform*. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 04/09/2015) (cit. on p. 4).

[10] Kevin Hammond and Greg Michelson, eds. *Research Directions in Parallel Functional Programming*. London, UK: Springer-Verlag, 1999. ISBN: 1-85233-092-9 (cit. on pp. 5, 9, 15, 134).

[11] Simon Marlow, editor. *Haskell 2010: Language Report*. 2010. URL: http://www.haskell.org/onlinereport/haskell2010 (visited on 12/09/2014) (cit. on pp. 6, 131).

[12] Simon Peyton Jones and John Hughes, eds. *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language*. Feb. 1999. URL: http://haskell.org/definition/original-haskell98-report.ps.gz (visited on 03/09/2015) (cit. on pp. 6, 131).

[13] Lennart Augustsson. *More Points for Lazy Evaluation*. URL: http://augustss.blogspot.com.es/2011/05/more-points-for-lazy-evaluation-in.html (visited on 11/09/2014) (cit. on pp. 6, 132).

[14] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl and Simon L. Peyton Jones. 'Algorithms + Strategy = Parallelism'. In: *Journal of Functional Programming* 8.1 (Jan. 1998), pp. 23–60 (cit. on pp. 7, 38, 58, 132).

[15] Simon Marlow et al. 'Seq No More: Better Strategies for Parallel Haskell'. In: *Proceedings of the 3rd ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, Sept. 2010, pp. 91–102 (cit. on pp. 8, 103, 133).

[16] GeorgeHoratiu Botorog and Herbert Kuchen. 'Efficient Parallel Programming with Algorithmic skeletons'. In: *Euro-Par'96 Parallel Processing*. Ed. by Luc Bougé, Pierre Fraigniaud, Anne Mignotte and Yves Robert. LNCS 1123. Springer Berlin Heidelberg, 1996, pp. 718–731 (cit. on p. 10).

[17] Patrick M. Sansom and Simon L. Peyton Jones. 'Time and Space Profiling for Non-Strict, Higher-Order Functional Languages'. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. New York, NY, USA: ACM, Jan. 1995, pp. 355–366 (cit. on pp. 11, 49, 136).

[18] Kevin Hammond, Hans-Wolfgang Loidl and Phil Trinder. 'Parallel Cost Centre Profiling'. In: *Proceedings of the 1997 Glasgow Workshop on Functional Programming*. Sept. 1997 (cit. on pp. 12, 58, 136).

[19] Kevin Hammond, Hans Wolfgang Loidl and Andrew Partridge. 'Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell'. In: *High Performance Functional Computing*. HPFC '95. Apr. 1995, pp. 208–221 (cit. on pp. 12, 57, 136).

[20] Don Jones Jr., Simon Marlow and Satnam Singh. 'Parallel Performance Tuning for Haskell'. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. New York, NY, USA: ACM, 2009, pp. 81–92 (cit. on pp. 12, 42, 59, 73, 137).

[21] Michiel Ronsse, Koen De Bosschere and Jacques Chassin de Kergommeaux. 'Execution Replay and Debugging'. In: *Proceedings of the 4th International Workshop on Automated Debugging*. AADEBUG 2000. Aug. 2000 (cit. on pp. 13, 20, 138).

[22] Frank Cornelis et al. '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 (cit. on pp. 13, 20, 138).

[23] Mubrak S. Mohsen, Rosni Abdullah and Yong M. Teo. 'A Survey on Performance Tools for OpenMP'. In: *World Academy of Science, Engineering and Technology* 49 (2009), pp. 754–765 (cit. on p. 17).

[24] Jeffrey Vetter and Chris Chambreau. *mpiP: Lightweight, Scalable MPI Profiling, Version 3.4.1*. Mar. 2014. URL: http://mpip.sourceforge.net (visited on 04/09/2015) (cit. on p. 18).

[25] *Vampir*. URL: http://www.vampir.eu (visited on 05/09/2015) (cit. on p. 18).

[26] W. E. Nagel et al. 'VAMPIR: Visualization and Analysis of MPI Resources'. In: *Supercomputer* 12.1 (1996), pp. 69–80 (cit. on p. 18).

[27] Xavier Aguilar Fruto. 'Towards Scalable Performance Analysis of MPI Parallel Applications'. Licenciate thesis. Sweden: KTH Royal Institute of Technology, 2015 (cit. on p. 18).

[28] *Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes*. URL: http://www.score-p.org (visited on 05/09/2015) (cit. on p. 18).

[29] Andreas Knüpfer et al. 'Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir'. In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel and Michael M. Resch. Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31475-9 (cit. on p. 18).

[30] *TAU – Tuning and Analysis Utilities*. URL: http://www.cs.uoregon.edu/research/tau/home.php (visited on 05/09/2015) (cit. on pp. 18, 19).

[31] *Scalasca*. URL: http://www.scalasca.org (visited on 07/09/2015) (cit. on p. 18).

[32] Markus Geimer et al. 'The Scalasca Performance Toolset Architecture'. In: *Concurrency and Computation: Practice and Experience* 22.6 (Apr. 2010), pp. 702–719 (cit. on p. 18).

[33] *ompP: OpenMP Profiler*. URL: http://ompp-tool.com (visited on 06/09/2015) (cit. on p. 18).

[34] Karl Fürlinger and Michael Gerndt. 'ompP: A Profiling Tool for OpenMP'. In: *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*. IWOMP '05/IWOMP '06. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 15–23 (cit. on p. 19).

[35] Sameer S. Shende and Allen D. Malony. 'The TAU Parallel Performance System'. In: *International Journal of High Performance Computing Applications* 20.2 (May 2006), pp. 287–311 (cit. on p. 19).

[36] Robert Bell, Allen D. Malony and Sameer Shende. 'ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis'. In: *Euro-Par 2003*. Ed. by Harald Kosch, László Böszörményi and Hermann Hellwagner. LNCS 2790. Springer Berlin Heidelberg, 2003, pp. 17–26. ISBN: 978-3-540-40788-1 (cit. on p. 19).

[37] Satish Narayanasamy. 'Deterministic Replay using Processor Support and Its Applications'. PhD thesis. United States: University of California, 2007 (cit. on p. 20).

[38] Bob Boothe. 'Efficient Algorithms for Bidirectional Debugging'. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, June 2000, pp. 299–310 (cit. on p. 20).

[39] Julian Seward and Nicholas Nethercote. 'Using Valgrind to Detect Undefined Value Errors with Bit-precision'. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA, USA: USENIX Association, 2005, pp. 17–30 (cit. on p. 20).

[40] João Pedro Marques Silva. 'Ditto – Deterministic Execution Replay for the Java Virtual Machine on Multi-processors'. MA thesis. Portugal: Universidade Técnica de Lisboa, Oct. 2012 (cit. on pp. 20, 21).

[41] Dennis Geels, Gautam Altekar, Scott Shenker and Ion Stoica. 'Replay Debugging for Distributed Applications'. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '06. Boston, MA, USA: USENIX Association, 2006, pp. 289–300 (cit. on p. 21).

[42] Thomas J. LeBlanc and John M. Mellor-Crummey. 'Debugging Parallel Programs with Instant Replay'. In: *IEEE Transactions on Computers* C-36 (4 Apr. 1987), pp. 471–482 (cit. on p. 21).

[43] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman and Peter M. Chen. 'Execution Replay of Multiprocessor Virtual Machines'. In: *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '08. Seattle, WA, USA: ACM, Mar. 2008, pp. 121–130 (cit. on p. 22).

[44] Gautam Altekar and Ion Stoica. 'ODR: Output-deterministic Replay for Multicore Debugging'. In: *Proceedings of the ACM SIGOPS*

22nd Symposium on Operating Systems Principles. SOSP '09. Big Sky, Montana, USA: ACM, Oct. 2009, pp. 193–206 (cit. on pp. 22, 126).

[45] Soyeon Park et al. 'PRES: Probabilistic Replay with Execution Sketching on Multiprocessors'. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, Oct. 2009, pp. 177–192 (cit. on pp. 22, 126).

[46] Jeff Huang, Peng Liu and Charles Zhang. 'LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs'. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 207–216 (cit. on p. 22).

[47] Simon L. Peyton Jones et al. 'The Glasgow Haskell Compiler: A Technical Overview'. In: *Proceedings of Joint Framework for Information Technology Technical Conference*. Mar. 1993, pp. 249–257 (cit. on p. 25).

[48] L. Augustsson and T. Johnsson. 'The Chalmers Lazy ML-Compiler'. In: *The Computer Journal* 32.2 (Apr. 1989), pp. 127–141 (cit. on pp. 25, 46).

[49] Paul Hudak, John Hughes, Simon Peyton Jones and Philip Wadler. 'A History of Haskell: Being Lazy with Class'. In: *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55 (cit. on p. 25).

[50] Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne. 'Concurrent Haskell'. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. New York, NY, USA: ACM, Jan. 1996, pp. 295–308 (cit. on pp. 25, 34).

[51] Simon Marlow, Simon Peyton Jones and Wolfgang Thaller. 'Extending the Haskell Foreign Function Interface with Concurrency'. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. New York, NY, USA: ACM, Sept. 2004, pp. 22–32 (cit. on pp. 25, 36).

[52] Tim Harris, Simon Marlow, Simon Peyton-Jones and Maurice Herlihy. 'Composable Memory Transactions'. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: ACM, June 2005, pp. 48–60 (cit. on p. 25).

[53] Tim Sheard and Simon Peyton Jones. 'Template Meta-programming for Haskell'. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, Oct. 2002, pp. 1–16 (cit. on p. 25).

[54] Mark P. Jones. 'A Theory of Qualified Types'. In: *Proceedings of the 4th European Symposium on Programming*. ESOP '92. Rennes, France: Springer-Verlag, Feb. 1992, pp. 287–306 (cit. on p. 25).

[55] Kung Chen, Paul Hudak and Martin Odersky. 'Parametric Type Classes'. In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA: ACM, Jan. 1992, pp. 170–181 (cit. on p. 25).

[56] Mark P. Jones. 'Type Classes with Functional Dependencies'. In: *Programming Languages and Systems. 9th European Symposium on Programming, ESOP 2000*. Ed. by Gert Smolka. Vol. 1782. LNCS 1782. Springer Berlin Heidelberg, 2000, pp. 230–244 (cit. on p. 25).

[57] Konstantin Läufer. 'Type classes with existential types'. In: *Journal of Functional Programming* 6.03 (May 1996), pp. 485–518 (cit. on p. 25).

[58] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Mark Shields. 'Practical Type Inference for Arbitrary-rank Types'. In: *Journal of Functional Programming* 17.1 (Jan. 2007), pp. 1–82 (cit. on p. 25).

[59] Simon Peyton Jones, Geoffrey Washburn and Stephanie Weirich. *Wobbly types: type inference for generalised algebraic data types*. Tech. rep. MS-CIS-05-26. Philadelphia, Pennsylvania: University of Pennsylvania, July 2004 (cit. on p. 25).

[60] Duncan Coutts, Isaac Potoczny-Jones and Don Stewart. 'Haskell: Batteries Included'. In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: ACM, Sept. 2008, pp. 125–126 (cit. on p. 25).

[61] Simon Marlow and Simon Peyton Jones. 'The Glasgow Haskell Compiler'. In: *The Architecture of Open Source Applications, Volume 2*. Ed. by Ami Brown and Greg Wilson. Licensed under a Creative Commons "Attribution 3.0 Unported" license. Apr. 2012, pp. 67–88. URL: http://aosabook.org/en/ghc.html (cit. on pp. 27, 28).

[62] Andrew Tolmach, Tim Chevalier and The GHC Team. 'An External Representation for the GHC Core Language (For GHC 6.10)'. July 2009. URL: https://downloads.haskell.org/~ghc/6.10.4/docs/html/ext-core/core.pdf (visited on 13/05/2015) (cit. on p. 27).

[63] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones and Kevin Donnelly. 'System F with Type Equality Coercions'. In: *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI '07. Nice, France: ACM, 2007, pp. 53–66 (cit. on p. 27).

[64] Simon Peyton Jones, Tony Hoare and Andrew Tolmach. 'Playing by the rules: rewriting as a practical optimisation technique'. In: *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*. Haskell '01. Sept. 2001, pp. 203–233 (cit. on p. 27).

[65] Simon Peyton Jones and Simon Marlow. 'Secrets of the Glasgow Haskell Compiler Inliner'. In: *Journal of Functional Programming* 12.4–5 (July 2002), pp. 393–434 (cit. on pp. 27, 45).

[66] Simon L. Peyton Jones. 'Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine'. In: *Journal of Functional Programming* 2.02 (Apr. 1992), pp. 127–202 (cit. on p. 27).

[67] Norman Ramsey, Simon Peyton Jones and Christian Lindig. 'The C– Language Specification. Version 2.0 (CVS Revision 1.128)'. Feb. 2005. (Visited on 13/05/2015) (cit. on p. 27).

[68] Norman Ramsey, João Dias and Simon Peyton Jones. 'Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation'. In: *Proceedings of the 3rd ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, Sept. 2010, pp. 121–134 (cit. on p. 27).

[69] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization*. CGO '04. Palo Alto, California, Mar. 2004 (cit. on p. 27).

[70] David A. Terei and Manuel M.T. Chakravarty. 'An LLVM Backend for GHC'. In: *Proceedings of the 3rd ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, Sept. 2010, pp. 109–120 (cit. on p. 27).

[71] Simon Marlow, Tim Harris, Roshan P. James and Simon Peyton Jones. 'Parallel generational-copying garbage collection with a block-structured heap'. In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM '08. Tucson, AZ, USA: ACM, June 2008, pp. 11–20 (cit. on p. 28).

[72] Andrew W. Appel. 'Simple generational garbage collection and fast allocation'. In: *Software: Practice and Experience* 19.2 (Feb. 1989), pp. 171–183 (cit. on p. 28).

[73] Richard Jones. 'Tail recursion without space leaks'. In: *Journal of Functional Programming* 2.01 (Jan. 1992), pp. 73–79 (cit. on pp. 33, 39).

[74] Simon Marlow, Alexey Rodriguez Yakushev and Simon Peyton Jones. 'Faster Laziness Using Dynamic Pointer Tagging'. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: ACM, Oct. 2007, pp. 277–288 (cit. on p. 34).

[75] Tim Harris, Simon Marlow and Simon Peyton Jones. 'Haskell on a Shared-Memory Multiprocessor'. In: *Proceedings of the 2005*

ACM SIGPLAN Workshop on Haskell. Haskell '05. ACM, Sept. 2005, pp. 49–61 (cit. on pp. 34, 39, 41).

[76] Marlow Marlow, Simon Peyton Jones and Satnam Singh. 'Runtime Support for Multicore Haskell'. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, Sept. 2009, pp. 65–78 (cit. on pp. 34, 36, 41, 103).

[77] Jost Berthold, Simon Marlow, Kevin Hammond and Abdallah Al Zain. 'Comparing and Optimising Parallel Haskell Implementations for Multicore Machines'. In: *Proceedings of the 38th International Conference on Parallel Processing Workshops*. ICPPW '09. Vienna, Austria: IEEE Computer Society, Sept. 2009, pp. 386–393 (cit. on p. 36).

[78] David Chase and Yossi Lev. 'Dynamic Circular Work-stealing Deque'. In: *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. New York, NY, USA: ACM, July 2005, pp. 21–28 (cit. on p. 39).

[79] Simon Marlow. 'Why can't I get a stack trace?' Talk at the ACM SIGPLAN Haskell Implementors' Workshop. Sept. 2012. URL: https://wiki.haskell.org/wikiupload/6/6c/Hiw2012-simon-marlow.pdf (cit. on p. 44).

[80] Guy Lewis Steele Jr. 'Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO'. In: *Proceedings of the 1977 Annual Conference*. ACM '77. Seattle, Washington: ACM, 1977, pp. 153–162 (cit. on p. 44).

[81] Colin Runciman and David Wakeling. 'Heap Profiling of Lazy Functional Programs'. In: *Journal of Functional Programming* 3.2 (Apr. 1993), pp. 217–245 (cit. on pp. 46, 49).

[82] Lennart Augustsson. *The HBC compiler*. URL: http://web.archive.org/web/20090815181116/http://www.cs.chalmers.se/~augustss/hbc/hbc.html (visited on 02/06/2015) (cit. on p. 46).

[83] Colin Runciman and Niklas Röjemo. 'New Dimensions in Heap Profiling'. In: *Journal of Functional Programming* 6.04 (July 1996), pp. 587–620 (cit. on pp. 48, 49).

[84] Niklas Röjemo. 'Highlights from nhc – A Space-Efficient Haskell Compiler'. In: *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. New York, NY, USA: ACM, 1995, pp. 282–292 (cit. on p. 48).

[85] Niklas Röjemo. 'Garbage Collection, and Memory Efficiency, in Lazy Functional Languages'. PhD thesis. Sweden: Chalmers University of Technology, May 1995 (cit. on p. 48).

[86] Patrick M. Sansom. 'Execution Profiling for Non-Strict Functional Languages'. PhD thesis. Scotland: University of Glasgow, Apr. 1994 (cit. on p. 49).

[87] R.G. Morgan and S.A. Jarvis. 'Profiling Large-Scale Lazy Functional Programs'. In: *Journal of Functional Programming* 8.3 (May 1998), pp. 201–237 (cit. on p. 50).

[88] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. New York, NY, USA: Cambridge University Press, Apr. 2003. ISBN: 978-0-521-82614-3 (cit. on p. 51).

[89] Henrik Nilsson. 'How to Look Busy While Being as Lazy as Ever: The Implementation of a Lazy Functional Debugger'. In: *Journal of Functional Programming* 11.6 (Nov. 2001), pp. 629–671 (cit. on p. 52).

[90] Henrik Nilsson. 'Declarative Debugging for Lazy Functional Languages'. PhD thesis. Sweden: Linköpings universitet, May 1998 (cit. on p. 52).

[91] Henrik Nilsson and Jan Sparud. 'The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging'. In: *Automated Software Engineering* 4.2 (Apr. 1997), pp. 121–150 (cit. on p. 52).

[92] Henrik Nilsson. 'Tracing Piece by Piece: Affordable Debugging for Lazy Functional Languages'. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*. ICFP '99. New York, NY, USA: ACM, Sept. 1999, pp. 36–47 (cit. on p. 52).

[93] Jan Sparud and Colin Runciman. 'Tracing Lazy Functional Computations Using Redex Trails'. In: *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP '97*. Ed. by Hugh Glaser, Pieter Hartel and Herbert Kuchen. LNCS 1292. Springer Berlin Heidelberg, Sept. 1997, pp. 291–308. ISBN: 978-3-540-63398-3 (cit. on p. 52).

[94] Jan Sparud and Colin Runciman. 'Complete and Partial Redex Trails of Functional Computations'. In: *Implementation of Functional Languages: 9th Interantional Workshop, IFL '97*. Ed. by Chris Clack, Kevin Hammond and Tony Davie. LNCS 1467. Springer Berlin Heidelberg, May 1998, pp. 160–177. ISBN: 978-3-540-64849-9 (cit. on p. 53).

[95] Olaf Chitil, Colin Runciman and Malcolm Wallace. 'Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs'. In: *Implementation of Functional Languages: 12th International Workshop, IFL 2000*. Ed. by Markus Mohnen and Pieter Koopman. LNCS 2011. Springer Berlin Heidelberg, Apr. 2001, pp. 176–193. ISBN: 978-3-540-41919-8 (cit. on p. 53).

[96] Malcolm Wallace, Olaf Chitil, Thorsten Brehm and Colin Runciman. 'Multiple-View Tracing for Haskell: a New Hat'. In: *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*. Haskell '01. Sept. 2001, pp. 151–170 (cit. on p. 53).

[97] Andy Gill. 'Debugging Haskell by Observing Intermediate Data Structures'. In: *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*. Haskell '00. Sept. 2000 (cit. on pp. 53, 54).

[98] Claus Reinke. 'GHood – Graphical Visualisation and Animation of Haskell Object Observations'. In: *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*. Haskell '01. Sept. 2001, pp. 121–149 (cit. on p. 54).

[99] Robert Ennals and Simon Peyton Jones. 'HsDebug: Debugging Lazy Programs by Not Being Lazy'. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell '03. New York, NY, USA: ACM, Aug. 2003, pp. 84–87 (cit. on p. 54).

[100] Robert Ennals and Simon Peyton Jones. 'Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs'. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Functional programming*. ICFP '03. New York, NY, USA: ACM, Aug. 2003, pp. 287–298 (cit. on p. 55).

[101] Paul Roe. 'Parallel Program Simulator via Transformation (A Tale of Parallel Quicksort)'. In: *Parallel Computing and Transputers*. PCAT '93. 1994, pp. 66–74 (cit. on p. 55).

[102] Paul Roe. 'Parallel Programming using Functional Languages'. PhD thesis. Scotland: University of Glasgow, Feb. 1991 (cit. on p. 56).

[103] Colin Runciman and David Wakeling. 'Profiling Parallel Functional Computations (Without Parallel Machines)'. In: *Functional Programming, Glasgow 1993*. Workshops in Computing. Springer London, 1994, pp. 236–251 (cit. on p. 56).

[104] Hans-Wolfgang Loidl. 'Granularity in Large-Scale Parallel Functional Programming'. PhD thesis. Scotland: University of Glasgow, Mar. 1998 (cit. on p. 57).

[105] David J. King, Jon Hall and Phil Trinder. 'A Strategic Profiler for Glasgow Parallel Haskell'. In: *Proceedings of the 10th International Workshop on the Implementation of Functional Languages*. IFL '98. Sept. 1998, pp. 88–102 (cit. on p. 58).

[106] Nathan Charles and Colin Runciman. 'An Interactive Approach to Profiling Parallel Functional Programs'. In: *Implementation of Functional Languages: 10th International Workshop, IFL '98*. Ed. by Kevin Hammond, Tony Davie and Chris Clack. LNCS 1595. Springer Berlin Heidelberg, Mar. 1999, pp. 20–37. ISBN: 978-3-540-66229-7 (cit. on p. 58).

[107] P. W. Trinder et al. 'GUM: A Portable Parallel Implementation of Haskell'. In: *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '96. New York, NY, USA: ACM, May 1996, pp. 79–88 (cit. on p. 58).

[108] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN: 013453333X (cit. on p. 78).

[109] Wiplove Mathur and Jeanine Cook. 'Toward Accurate Performance Evaluation Using Hardware Counters'. In: *Proceedings of the Applications for a Changing World, ITEA Modeling & Simulation Workshop*. Dec. 2003 (cit. on p. 91).

[110] Dmitrijs Zaparanuks, Milan Jovic and Matthias Hauswirth. 'Accuracy of Performance Counter Measurements'. In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software 2009*. ISPASS '09. IEEE, Apr. 2009, pp. 23–32 (cit. on p. 91).

[111] Will Partain. 'The nofib Benchmark Suite of Haskell Programs'. In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Ayr, Scotland: Springer London, July 1992, pp. 195–202 (cit. on p. 103).

[112] Henrique Ferreiro, Vladimir Janjic, Kevin Hammond and Laura Castro. 'Kindergarten Cop: Dynamic Nursery Resizing for GHC'. In: *Draft proceedings of the 25th Symposium on Implementation and Application of Functional Languages*. IFL '13. Aug. 2013 (cit. on p. 128).

INDEX