Pós-Graduação em Ciência da Computação

Luís Gabriel Nunes Ferreira Lima

UNDERSTANDING THE ENERGY BEHAVIOR OF CONCURRENT HASKELL PROGRAMS

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
<www.cin.ufpe.br/~posgraduacao>

RECIFE
2016

Luís Gabriel Nunes Ferreira Lima

UNDERSTANDING THE ENERGY BEHAVIOR OF CONCURRENT HASKELL PROGRAMS

A M.Sc. Dissertation presented to the Informatics Center of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Fernando José Castor de Lima Filho*
Co-Advisor: *João Paulo de Sousa Ferreira Fernandes*

RECIFE
2016

Catalogação na fonte
Bibliotecária Joana D'Arc Leão Salvador CRB 4-572

L732u Lima, Luís Gabriel Nunes Ferreira.
Understanding the energy Behavior of concurrent haskell programs / Luís Gabriel Nunes Ferreira Lima. – 2016.
76 f.: fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, Recife, 2016.
Inclui referências e apêndice.

1. Engenharia de software. 2. Programação funcional. 3. Programação concorrente. 4. Consumo de energia. I. Lima Filho, Fernando José Castor de (Orientador). II. Titulo.

005.1 CDD (22. ed.) UFPE-MEI 2016-112

Luís Gabriel Nunes Ferreira Lima

Understanding the Energy Behavior of Concurrent Haskell Program

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 22/08/2016

To my parents.

Acknowledgements

This work would not have been possible without the support of many. I would like to thank and dedicate this dissertation to the following people:

To my advisor Fernando Castor. Castor is an exceptional professor. His guidance and enthusiasm were fundamental to motivate me throughout this research.

To my co-advisor João Paulo Fernandes for his pivotal contributions to the work we have done.

To Paulo Lieuthier, Francisco Soares-Neto, and Gilberto Melfe for their numerous direct contributions to this dissertation.

To my friends, for being a part of who I am today. Special thanks go to Lucas, Tiago, Cláudio, Gustavo, Marcela, and Marina.

To my friends and colleagues from INDT. Thank you for the thought-provoking discussions, endless laughs, and beers.

To my family, especially my parents and my sister, for always supporting me through my whole life. I would not be here without them.

Life's barely long enough to get good at one thing.
So be careful what you get good at.
—RUSTIN COHLE (True Detective: After You've Gone)

Resumo

Há anos eficiência energética é uma preocupação para designers de hardware e software baixo-nível. Entretanto, a rápida proliferação de dispositivos móveis alimentados por bateria combinado com o crescente movimento global em busca de sustentabilidade tem motivado desenvolvedores e pesquisadores a estudar o impacto energético de softwares de aplicação em execução. Trabalhos recentes tem estudado o efeito que fatores como obsfucação de código, refatorações em linguagem orientadas à objetos e tipos de dados tem em eficiência energética. Este trabalho tenta lançar luz sobre o comportamento energético de programas concorrentes escritos em uma linguagem puramente funcional, Haskell. Nós conduzimos um estudo empírico para avaliar o desempenho e o comportamento energético de três diferentes abordagens para gerenciamento de threads e três primitivas para controle de concorrência usando nove diferentes benchmarks com um espaço de exploração experimental de mais de 400 configurações. Neste estudo, descobrimos que pequenas mudanças podem fazer uma grande diferença em termos de consumo de energia. Por exemplo, em um dos benchmarks, sob uma configuração específica, escolher uma primitiva de controle de concorrência (MVar) ao invés de outra (TMVar) pode acarretar em uma economia de 60% em consumo de energia. Percebemos também que nem sempre a relação entre consumo de energia e desempenho é clara. Em alguns cenários analisados, a configuração com melhor desempenho também apresentou o pior consumo de energia. Para ajudar desenvolvedores a entender melhor essa complexa relação, nós estendemos duas ferramentas de análise de desempenho existentes para coletar e apresentar dados sobre consumo de energia. Adicionalmente, baseado nos resultados do nosso estudo empírico, listamos um conjunto de recomendações para desenvolvedores com boas práticas de como escrever código energeticamente eficiente nesse ambiente.

Palavras-chave: Eficiência Energética. Consumo de Energia. Haskell. Programação Concorrente. Programação Funcional. Análise de Desempenho.

Abstract

Energy-efficiency has concerned hardware and low-level software designers for years. However, the rapid proliferation of battery-powered mobile devices combined with the growing worldwide movement towards sustainability have caused developers and researchers to study the energy impact of application software in execution. Recent work has studied the effect that factors such as code obfuscation, object-oriented refactorings, and data types have on energy efficiency. In this work, we attempt to shed light on the energy behavior of concurrent programs written in a purely functional language, Haskell. We conducted an empirical study to assess the performance and energy behavior of three different thread management approaches and three primitives for concurrency control using nine different benchmarks with an experimental space exploration of more than 400 configurations. In this study, we found out that small changes can make a big difference in terms of energy consumption. For instance, in one of our benchmarks, under a specific configuration, choosing one concurrency control primitive (MVar) over another (TMVar) can yield 60% energy savings. Also, the relationship between energy consumption and performance is not always clear. We found scenarios where the configuration with the best performance also exhibited the worst energy consumption. To support developers in better understanding this complex relationship, we have extended two existing performance analysis tools also to collect and present data about energy consumption. In addition, based on the results of our empirical study, we provide a list of guidelines for developers with good practices for writing energy-efficient code in this environment.

Keywords: Energy-Efficiency. Energy Consumption. Haskell. Concurrent Programming. Functional Programming. Performance Analysis.

List of Figures

List of Tables

List of Codes

List of Acronyms

CSP	Communicating Sequential Processes	24
TM	Transactional Memory	24
STM	Software Transactional Memory	25
GHC	Glasgow Haskell Compiler	25
RTS	GHC runtime system	25
HEC	Haskell Execution Context	25
API	Application Programming Interface	27
CLBG	The Computer Language Benchmarks Game	42
WAI	Haskell Web Application Interface	43
RAPL	Running Average Power Limit	30
MSR	Machine-Specific Register	31
CAF	Constant Applicative Form	33
OLS	Ordinary Least Squares	37

Contents

1 Introduction

Um passo a frente e você não está mais no mesmo lugar
One step forward and you are not in the same place
—CHICO SCIENCE & NAÇÃO ZUMBI (Um Passeio No Mundo Livre)

The evolution of technology is unveiling a reality that could hardly be imaginable some decades ago. Nowadays, we can buy a multiprocessor computer with a few gigabytes of memory with the size of a credit card for less than a hundred dollars. The ability to manufacture such small and potent devices is leading to a rapid proliferation of a variety of mobile computing platforms. These devices are part of a diverse ecosystem that includes smartwatches, smartphones, tablets, IoT sensors, and drones. As they grow in popularity, new challenges arise for the development community. *Energy consumption* is one of them. It is imperative for delivering a good user experience that these devices stay up and running for as long as possible. As battery lifetime is closely related to energy consumption, it means that building energy-efficient systems is becoming mandatory for providing value to the end user.

This concern, however, goes beyond unwired devices. On the other side of the spectrum, big internet companies are also affected by low energy efficiency. To deliver fast access to their services globally, these companies usually have to maintain a huge server infrastructure to host their products. In this kind of environment, due to its scale, the energy consumption can have a high impact on the maintenance costs. For instance, Facebook is building data centers inside the Arctic Circle in order to improve the system's energy efficiency by reducing the power needed for cooling[1]. This is just one example of the economic impact driven by the search for more efficient solutions. It shows that energy efficiency is becoming a key design attribute for building computational systems.

Although it may seem like a recent problem, the energy efficiency of computer systems has been a concern for researchers for a long time. Initially, most of the research focused on the hardware design layer, developing new ways to build electronic components that wasted

[1]http://www.bloomberg.com/news/articles/2013-10-04/facebooks-new-data-center-in-sweden-puts-the-heat-on-hardware-makers

less energy (CHANDRAKASAN; SHENG; BRODERSEN, 1992). This was motivated by the assumption that only hardware dissipates power, not software. However, in a computer system, software plays a fundamental role in deciding how a computational task will be executed on specific hardware. For this reason, software can have a substantial impact on energy consumption.

From a software perspective, the energy efficiency problem can be tackled at different levels of abstraction, ranging from machine code level to user-facing applications. Traditionally, the research in this area has been focused on low-level software. Much progress has been achieved on building energy-efficient solutions for embedded software (TIWARI; MALIK; WOLFE, 1994), compilers (HSU; KREMER, 2003), operating systems (MERKEL; BELLOSA, 2006) and runtime systems (RIBIC; LIU, 2014; FARKAS et al., 2000). However, the growing worldwide movement towards sustainability, including sustainability in software (BECKER et al., 2015), has motivated the study of the energy impact of application software in execution.

Recent empirical studies have provided initial evidence that high-level decisions can effectively reduce the energy usage of application software (HINDLE, 2012; TREFETHEN; THIYAGALINGAM, 2013; PINTO; CASTOR; LIU, 2014b; SAHIN; POLLOCK; CLAUSE, 2014). A big advantage of this kind of optimization is that it is complementary to the low-level ones, which helps improving the energy efficiency of the system as a whole. Also, it includes the developer in the loop of deciding how to optimize its software for a certain context. As they hold the knowledge about the application domain, this can lead to more aggressive optimizations. In contrast, low-level optimizations have to be usually more generic and, consequently, more conservative.

Nevertheless, the programming models used for developing software are an important piece of this puzzle when we talk about writing energy-efficient software. A trend that we can identify in modern software development is the employment of concurrency techniques as a way improve the program's performance. The main reason for this is the increasing popularity of multicore processors. To leverage this kind of architecture, developers often have to create multiple flows of control in their programs and manually orchestrate them to guarantee correctness. This is, however, a difficult and error-prone task to accomplish (SUTTER; LARUS, 2005; HERLIHY; SHAVIT, 2012).

A commonly referred alternative to mitigate this problem is the use of functional programming. Several functional programming languages such as Clojure, Erlang, Elixir, and Scala have gained popularity in the last decade emphasizing features that supposedly makes it easier dealing with concurrency. Features such as immutable data structures, Actor Model, and Software Transactional Memory are a few of them. Haskell, in this context, is a language that stands out for having many different abstractions for parallel and concurrent programming.

Finally, the relationship between energy consumption and concurrency is still uncertain. We have a vast literature on how to improve the performance of programs by employing concurrency, but we still do not understand with the same depth how these techniques impact energy consumption. Depending on the scenario, the gains in performance may not be worth the decrease in energy efficiency. Recent studies have shed light on this matter for the concurrency toolkit of the Java programming language (PINTO; CASTOR; LIU, 2014b; PINTO et al., 2016). In the functional programming world, however, this is a subject yet to be explored.

In this work, we believe that functional programming can play a major role in helping developers to write correct concurrent programs. Moreover, we think that educating developers and providing the necessary tools for them to write energy-efficient code in this environment is crucial for making sure they develop software that meets the usability and business requirements of the modern world. To achieve this goal, we need to understand how high-level decisions change the behavior of a program regarding its performance and energy consumption.

1.1 Problem

In this work, we tackle two critical problems in the development of energy-efficient concurrent applications. The first one is the *lack of knowledge*. A recent study by Pinto, Castor and Liu (2014a) shows that, although application developers are consistently more interested in understanding how to reduce energy consumption in their software, there is a general lack of information in the community about how it can be achieved. This is unfortunate because it is very unlikely that developers will be able to build energy-efficient solutions without a logical and systematical way of reasoning about the energy consumption of the software that they write.

The second problem is the *lack of tools*. Currently, there is not much tooling in which developers can rely on to analyze the energy consumption of their programs. Most of the tools available are closer to the systems side than the software side, which makes it difficult to establish a relation with the software in execution. This problem is closely related to the previous one as the availability of such tools can lead to a better understanding about software energy consumption in general.

1.2 Goal

The goal of this work is to mitigate both aforementioned problems: the lack of knowledge, and the lack of tools for developing energy-efficient concurrent applications. Moreover, we are interested in tackling these problems under the functional programming point-of-view, using Haskell as our platform. To achieve this goal, we investigate the following key research questions:

RQ1. How can we effectively measure the energy consumption of a Haskell application?

RQ2. Do alternative concurrent constructs have different impacts on energy consumption?

RQ3. How can developers improve the energy efficiency of their concurrent applications?

To answer **RQ1**, we studied two performance analysis tools of the Haskell ecosystem in order to understand how they could be extended to measure the energy consumption of a Haskell program. To answer **RQ2**, we conducted an empirical study aiming to illuminate the relationship between the choices and settings of six concurrent programming constructs available in Haskell and energy consumption. In these six, we have three thread management constructs (`forkIO`, `forkOn`, and `forkOS`) and three primitives for sharing data (`MVar`, `TMVar`, and `TVar`). This study consisted of an extensive experimental space exploration over both microbenchmarks and real-world Haskell programs, which were manually modified to use each of constructs that we described. Finally, to answer **RQ3**, we elaborated a list of guidelines based on the results of our empirical study for educating developers on how to improve the energy consumption of their concurrent applications.

1.3 Contributions

This work sheds light on the energy behavior of Haskell programs. To the best of our knowledge, this is the first attempt to analyze energy efficiency in the context of functional programming languages. Moreover, this work makes the following contributions:

- **A tool for fine-grained energy analysis.** We extend the GHC profiler to collect and report fine-grained information about the energy consumption of a Haskell program;
- **A tool for coarse-grained energy analysis.** We extend the Criterion microbenchmarking library to collect, perform and report statistical performance analysis of the energy consumption of Haskell code;
- **An understanding of the energy behavior of concurrent Haskell programs.** We conduct an extensive experimental space exploration illuminating the relationship between the choices and settings of Haskell's concurrent programming constructs, and performance and energy consumption over both microbenchmarks and real-world Haskell programs;
- **A list of guidelines on how to write energy-efficient software in Haskell.** We provide some recommendations for helping software developers to improve the energy efficiency of their concurrent Haskell programs.

1.4 Outline

The remainder of this work is structured as follows:

- **Chapter 2** reviews essential concepts used throughout this work. First, we briefly introduce the Haskell programming language. We show through a series of code samples some important and distinct features of the language. Second, we present an overview of the fundamentals of concurrent programming. Finally, we present how Haskell approaches concurrent programming. We show which abstractions the language provides for both creating new threads of execution and sharing data among these threads. We also present an overview of how the Haskell's runtime system handles threading on multiprocessors;

- **Chapter 3** shows how to measure the energy consumption of Haskell programs. First, we explain what is RAPL and how it can be used to collect energy data. Then, we present in details two performance analysis tools of the Haskell ecosystem that we extended to also work with the energy consumption metrics;
- **Chapter 4** shows how different concurrent constructs impact energy consumption. We present an empirical study that we conducted considering three distinct thread management constructs and primitives for sharing data. Through an extensive experimental space exploration over microbenchmarks and real-world Haskell programs, we produce a list of findings about the energy behaviors of concurrent Haskell programs;
- **Chapter 5** presents a list of recommendations for Haskell developers on how to write energy-efficient software. These recommendations are based on the results of our empirical study;
- **Chapter 6** discusses previous research related to this work;
- **Chapter 7** present our concluding remarks and discuss where this work might lead.

In its essence, the content presented in **Chapter 4** has been one of the core contributions of a paper that has been published at the main research track of the *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)* (LIMA et al., 2016). When writing this dissertation, this chapter was extended and revised. A new version of this paper including these extensions and the remainder of this dissertation is under work to be submitted to a software engineering journal.

2 Background

Sometimes, the elegant implementation is just a function.
Not a method. Not a class. Not a framework.
Just a function.
—JOHN CARMACK

In this chapter, we review and introduce some essential concepts used in this work. First, we give a short introduction to the Haskell Programming Language in Section 2.1. Later, we briefly describe the fundamentals of concurrency in Section 2.2. Finally, we present the primitives and constructors for concurrent programming in Haskell in Section 2.3.

2.1 Haskell

Haskell is a *purely functional* programming language. Being functional means that functions are the building blocks of programs written in this language. Being pure means that no side-effect happens when evaluating a function. These two characteristics together make Haskell fundamentally different from imperative programming languages. In imperative programming languages, a program is expressed as a sequence of instructions that mutate data. In Haskell, a program is expressed as a composition of expressions where all state is controlled by passing arguments to function calls and returning values from them. Also, having no side-effect guarantees that, in a given execution context, a function executed with a given argument will always produce the same result. This property is known as *referential transparency*. It enforces that a program's behavior cannot depend on history, which improves reasoning about programs.

Haskell is also a *lazy* programming language. Lazy refers to a non-strict evaluation strategy also known as *call-by-need*. This strategy delays the evaluation of an expression until its value is needed. It avoids repeated evaluations, which can lead to performance improvements. This strategy also makes it possible to construct potentially infinite data structures.

Regarding programming style, *recursion* is the norm in Haskell since regular iterative loops require state mutation. To make it easier to express recursive functions, Haskell also has *pattern matching*. In Code 2.1, we can see an example of a recursive function using pattern matching. Line 1 is the base case, when the `factorial` receives zero as an argument, and Line 2 is the general case.

Code 2.1: A recursive factorial function

```
1 factorial :: Int -> Int
2 factorial 0 = 1
3 factorial x = x * factorial (x - 1)
```

Functions in Haskell can be *polymorphic*, which means that a function can be generalized to work with multiple types instead of a single one. Other programming languages have similar features such as *generics* in Java and *templates* in C++. Code 2.2 shows an example of a polymorphic function that can reverse a list of any type. In this example, `t` is a *parametric type* used to bind the input type to the output type of `reverse`. It reads: `reverse` receives a list of any type and returns a list of the same type as the input. This kind of polymorphism is known as *parametric polymorphism* (CARDELLI; WEGNER, 1985).

Code 2.2: A polymorphic function to reverse a list

```
reverse :: [t] -> [t]
reverse l = rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

Another characteristic of Haskell is that functions are values. This means that a function can receive other functions as arguments, and can also be the result of a function evaluation. This feature is known as *high-order functions*. It enables very popular functional patterns such as `map`, `filter`, and `reduce`. In Code 2.3, we can see `map` implemented in Haskell. It returns the list obtained by applying the provided function to each element of the input list.

Code 2.3: The `map` function

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

In Haskell, a developer can also extend the built-in primitive types by defining new *abstract data types*. Code 2.4 shows an example defining the `Tree` data type and the function `depth` to calculate the depth of the tree. As this example shows, the parametric polymorphism also works for abstract data types. Here, `Tree` has two constructors `Node` and `Empty`, where the first one holds three elements: the value of the node, the left, and right sub-trees. Pattern matching can be used to walk through a `Tree` as shown in Lines 5 and 6.

Code 2.4: A data type for binary tree and a function to calculate its depth

```haskell
1 data Tree t = Node t (Tree t) (Tree t)
2             | Empty
3
4 depth :: Tree t -> Int
5 depth Empty        = 0
6 depth (Node _ l r) = 1 + max (depth l) (depth r)
```

There is also a concept called *type classes* that enhances the definition of new types in Haskell. A type class is similar to an interface in Java. It defines a set of functions that can be applied to a particular type. This set of functions can be seen as a protocol to which a type must comply. So to instantiate a type class, a type must have its own implementation of each function defined by the protocol. As interfaces in Java, type classes were designed as a way for implementing ad hoc polymorphism in Haskell. The main difference between the two concepts is that type class instances are declared separately from the declaration of the corresponding types; while in Java, the definition of a class must declare any interfaces it implements. In Code 2.5, we show an example of instantiation of the `Eq` type class for the `Tree` type defined earlier. It states that a `Tree` is comparable for equality if its contained type is also comparable for equality.

Code 2.5: Definition of an `Eq` typeclass instance for the `Tree` data type

```haskell
instance Eq a => Eq (Tree a) where
  (==) Empty Empty = True
  (==) Empty (Node _ _ _) = False
  (==) (Node _ _ _) Empty = False
  (==) (Node x xl xr) (Node y yl yr) = (x == y) && (xl == yl) && (xr == yr)
```

The `Monad` typeclass is particularly important for Haskell because it allow developers to emulate mutable behavior and side-effects in a purely functional manner. Its definition can be seen in Code 2.6. The most important functions of this interface are `(>>=)`, also known as *bind*, and `return`, also called *unit*. The first one binds the contained value of the monad `m` to the parameter of its argument function. The second one wraps a value in the monad `m` and returns it. It is a common idiom to call a *monad* any type that is an instance of class `Monad`.

Two monads are particularly important for this work: `IO` and `STM`. The first one defines an environment to execute input/output operations. The second defines an environment for Software Transactional Memory, which is presented in Section 2.3. In Code 2.7, we have a basic example of how to perform I/O in Haskell. For instance, the functions `putStrLn` and `getLine` from the module `System.IO` print and read a `String` from the standard I/O,

Code 2.6: Definition of type class `Monad`

```haskell
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

respectively. The result type of `main` is `IO()`, where `()` is an empty tuple value. It represents that no value is returned, analogous to the type `void` on imperative languages.

Code 2.7: Basic `IO` example

```haskell
-- Using Monad operators
main :: IO ()
main = putStrLn "What is your name?" >> getLine
       >>= \name -> putStrLn ("Hey " ++ name ++ ", you rock!")

-- Using do-notation
main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

Finally, as you can see in Code 2.7, there is a notation in Haskell that makes it easier to express operation within a monad, the *do-notation*. Using `do`, a series of monadic function calls is sequenced as if in an imperative program. It works mainly as syntactic sugar for `(>>=)` and `(>>)` calls, binding variables that later become arguments of other functions and mainly sequentially composing the calls.

2.2 Concurrency

Concurrency and concurrent programming are on the rise nowadays due to the proliferation of multicore processors. However, these concepts are present in computer science since the early 1970s with the introduction of *time-sharing* (LEA, 2006). This model enables, for example, multi-tasking, which allows users to do several things at the same time in a computer such as browsing the Internet, playing music, and writing a document. To make this possible, the operating system scheduler has to share the processor time between all other processes that want to use this resource. By doing so, the user feels like the programs are executing at the same time although the processor is actually executing each program in a different time slice.

Concurrency is also usually associated (sometimes indistinctly) to parallelism. Although they are similar concepts, they are not the same. Concurrency consists in logically structuring programs in distinct control flows. The execution of these control flows is interlaced to simulate simultaneity when the underlying processor has only one core. On the other hand, parallelism is concerned with improving a program's performance by executing several computations in parallel, which requires a multicore processor. Depending on the number of cores available, the execution of a program can be literally parallel, entirely time-shared, or a combination of both.

In operating systems, there are two well-known concurrent programming abstractions to express an alternative flow of control: *processes* and *threads*. A process is a self-contained execution environment that holds all the information needed to run a program. Creating a new process is an onerous operation due to the significant number of resources it requires such as memory, registers, and address space. For this reason, they are known to be *heavyweight*. On the other hand, threads are the smallest concurrency unit in modern operating systems (TANENBAUM, 2007). As threads are contained within a process, there is a low overhead associated with creating new threads because they all share the same memory and address space. Due to this fact, threads are known to be *lightweight*. For instance, creating a thread is around 100 times faster than creating a process in POSIX systems (BUTENHOF, 1997).

In high-level programming languages, threads are the weapon of choice for concurrent programming. When compared to processes, they are faster and easier to manage. However, creating new threads by itself is not enough for building complex concurrent systems. We need mechanisms to enable coordination among the different flows of control. There are two mainstream communication strategies that enable cooperation between threads. The first one is sharing memory. In this approach, threads communicate with each other through reads and writes to a common memory location. To ensure program consistency, they need to control the access to these common locations. This control is commonly achieved through different synchronization mechanisms such as semaphores and mutexes. In these strategies, a thread has to acquire a lock as a way to communicate its access to a resource. However, this method is very prone to concurrency hazards such as deadlocks and livelocks (HERLIHY; SHAVIT, 2012).

The other communication mechanism is message passing. In this approach, the components communicate by exchanging messages. Each component (or process) is isolated, which means that there is no memory sharing. Two particular styles of message passing are popular: the Actor Model (AGHA, 1986) and Communicating Sequential Processes (CSP) (HOARE, 1978). The main difference is that the first is *asynchronous* while the second is *synchronous*. In CSP, a process that is sending a message blocks until the receiver accepts it. In the Actor Model, the messages are kept in the receiver's mailbox to avoid blocking the sender. The Actor Model is very popular in functional programming languages such as Erlang (ARMSTRONG, 2007) and Scala (HALLER; ODERSKY, 2009). The CSP model serves as inspiration for the concurrency abstractions of the Go programming language (PIKE, 2012).

Another mechanism to consistently coordinate concurrent threads in a shared memory scenario is Transactional Memory (TM). The basic idea is very simple. The runtime should take care of controlling the access to common memory locations. It uses an abstraction called *transaction* that behaves similarly to database transactions. In this approach, code that access shared memory is wrapped inside a transaction. Any conflicts that occur when threads are concurrently accessing the same location activate recovery strategies. These strategies ensure that each transaction is executed as if atomically and in isolation (with no intermediary state visible to other threads).

The first time this idea of using an abstraction such as database transactions to ensure consistency of shared data was presented by Lomet (1977). It was then formalized as Transactional Memory by Herlihy and Moss (1993). They proposed a hardware-supported transactional memory as a mechanism for building lock-free data structures. Although the original proposal required specialized hardware, we can now see implementations of Software Transactional Memory (STM) (SHAVIT; TOUITOU, 1995). There are several distinct implementations for different programming languages, including C/C++, Clojure, Java, Scala, and Haskell. Both Clojure and Haskell have STM support built into the core language. We will provide more details on Haskell's STM in the next section.

There is a fundamental difference between the two approaches for consistent data sharing. While locking strategies such as semaphores and mutexes try to avoid conflicts by not allowing concurrent access to shared data, STM assumes that no conflict will happen and, in case it happens, some action is taken to rollback the affected state. For this reason, the former is called *pessimistic concurrency* whereas the latter is called *optimistic concurrency*.

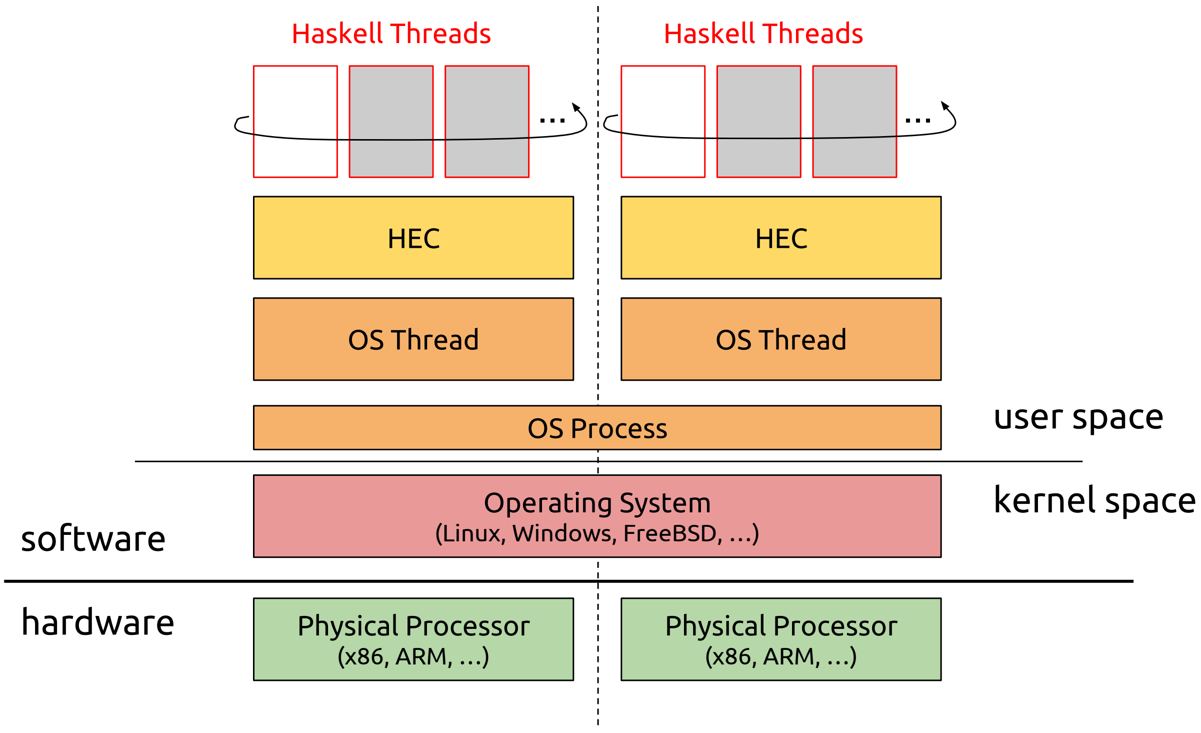
2.3 Concurrency in Haskell

The main component of the Haskell ecosystem is the Glasgow Haskell Compiler (GHC) (PEYTON-JONES et al., 1993). GHC provides a complete infrastructure for building, running, debugging, and profiling Haskell programs. It is composed of two core pieces: the compiler itself and the runtime system. The compiler translates source language into assembly code executable by a native host. The runtime system is a support library for primitive language services such as memory management and IO. As we will see in this section, concurrency in Haskell is enabled by a combination of both high-level constructs on the source language and a low-level infrastructure that is part of the GHC runtime system (RTS) (LI et al., 2007).

Haskell threads, also known as *green threads* (MARLOW, 2012), are the main abstraction for concurrent programming in Haskell. These are special threads managed by the GHC runtime system. They are multiplexed over a much smaller number of operating system threads. The RTS takes care of scheduling green threads to execute on a set of *virtual processors*. These virtual processors are also known as Haskell Execution Contexts (HECs) or *capabilities*. Each one can run one Haskell thread at a time. Each capability also has a run queue for keeping the Haskell threads that will run next.

The runtime system has an internal scheduler to manage the green threads. It uses a round-robin scheduling policy to manage the capabilities' run queue. So, each thread in the queue runs for a time slice[1] before being interrupted to the next one to run. The scheduler also performs load-balancing of Haskell threads. It moves threads from a capability's run queue to another to avoid CPU idle time[2]. These features make Haskell threads considerably more lightweight than regular OS threads. The GHC documentation states that: *"Typically Haskell threads are an order of magnitude or two more efficient (in terms of both time and space) than operating system threads."*[3] Another advantage of Haskell threads is the low context-switching overhead when comparing to OS threads. This is crucial for some systems with a high performance requirements such as web servers (VOELLMY et al., 2013).

Figure 2.1: Layers of concurrency in a Haskell stack



Source: Made by the author. Inspired on slides from the "GHC illustrated" presentation (TANI, 2015)

In Figure 2.1, we show the various layers involved in a concurrent system written in Haskell. In this example, the underlying machine has two cores. For this reason, there are two HECs associated with two OS threads. Although depicted this way, a developer can configure it differently. For instance, the number of capabilities can be set when the program is executed by passing a command line argument for the RTS (in this case, `-N`). Also, it is important to note that a program in Haskell can be linked with either threaded RTS or non-threaded RTS. The program should be linked to the threaded runtime to leverage multicore processors (as in Figure 2.1).[4]

[1]The default rescheduling time is 20ms. The developer can change it by passing a different value to the RTS via the `-i` command line argument

[2]The strategy used to move work from one capability to another is currently fixed, but there are some work towards making this policy customizable (SIVARAMAKRISHNAN et al., 2014)

[3]http://hackage.haskell.org/package/base-4.8.2.0/docs/Control-Concurrent.html#g:11

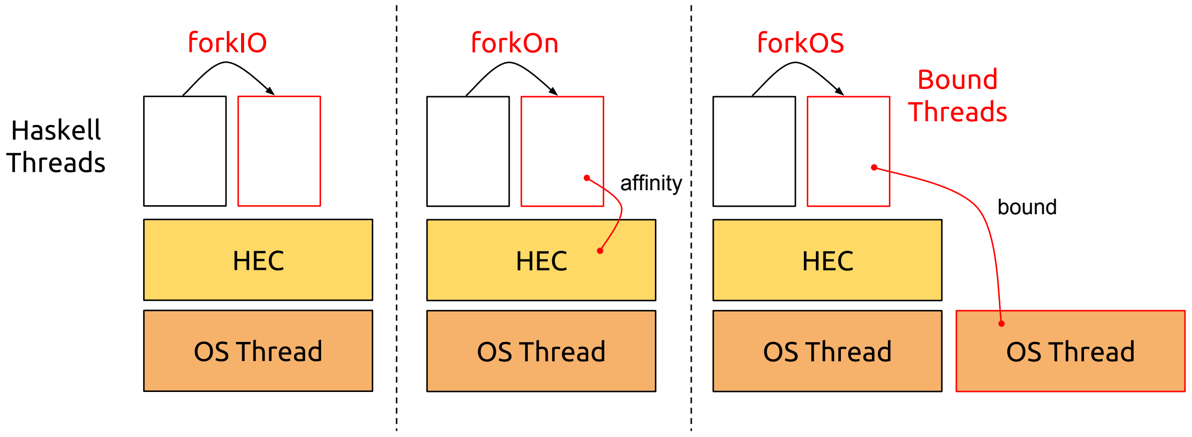
[4]This can be achieved by passing `-threaded` to GHC when compiling the program

Code 2.8: The thread creation interface

```haskell
forkIO :: IO () -> IO ThreadId
forkOn :: Int -> IO () -> IO ThreadId
forkOS :: IO () -> IO ThreadId
```

From a higher-level perspective, the original framework for concurrency in Haskell is called *Concurrent Haskell* (PEYTON-JONES; GORDON; FINNE, 1996). It represents a thread as a computation in the IO monad. We have three main functions to create a new thread in Haskell as Code 2.8 shows. `forkIO` spawns a Haskell thread. It takes an IO computation to be executed concurrently and returns a pointer to the newly created thread. `forkOn` also spawns a Haskell thread but lets the developer specify on which capability the thread should run. Unlike threads created with `forkIO`, the scheduler cannot migrate threads created with `forkOn` from one capability to another. For instance, if a program creates all its threads using `forkOn`, the scheduler will not be able to perform load-balancing. Finally, `forkOS` spawns a *bound thread*. A bound thread is a Haskell thread that is bound to an specific OS thread. They are treated by the scheduler the same way as other Haskell threads. The only difference is when it is time to run a bound thread. The capability has to run it on its bound OS thread. Figure 2.2 shows the difference between each thread creation function.

Figure 2.2: Thread creation functions



Source: Made by the author. Inspired on slides from the "GHC illustrated" presentation (TANI, 2015)

The basic concurrency control primitive of Haskell is the `MVar`. A value of type `MVar t` is a mutable location that is either empty or contains a value of type `t`. Code 2.9 shows the Application Programming Interface (API) for manipulating an `MVar`. The more commonly used functions are `newMVar`, `newEmptyMVar`, `takeMVar` and `putMVar`. The first one creates an `MVar` with a value inside. The second creates an empty `MVar`. The `takeMVar` function takes a value from an `MVar`, returning it in the IO monad. The operation returns immediatly if the `MVar` is full. For empty ones it will block until they are filled. The opposite applies to `putMVar`. The function puts a value in an `MVar`. It will return immediatly if the `MVar` is empty. Otherwise, the function will block until the `MVar` is emptied. An `MVar` combines both locking and condition-based synchronization in a single primitive. It also has a formally defined semantics (PEYTON-JONES; GORDON; FINNE, 1996).

Code 2.9: The `MVar` interface

```haskell
-- Type definition
data MVar a

-- MVar manipulation
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
tryTakeMVar  :: MVar a -> IO (Maybe a)
tryPutMVar   :: MVar a -> a -> IO Bool
isEmptyMVar  :: MVar a -> IO Bool

-- MVar creation
newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
```

In Haskell, we can also use software transactional memory to coordinate concurrency. Its implementation is called STM Haskell (HARRIS et al., 2005). Code 2.10 shows the API for STM in Haskell. The transactional variable, or `TVar`, is the main abstraction of this framework. A `TVar t` is a mutable location that holds a value of type `t`. Reads and writes to a `TVar` can only be accomplished inside the `STM` monad. To execute an `STM` monad we have to use the `atomically` function. This function runs an `STM` monad as a transaction, which makes a sequence of operations to take place atomically with respect to the rest of the program. The `atomically` function also acts as a bridge between the `IO` and `STM` monads, which allows `STM` operations to be executed by threads. It is important to note that, as `STM` and `IO` are different monads, Haskell's type system does not allow manipulating a `TVar` outside a transaction, nor does it allow performing an IO operation inside a transaction. This is an important property that is hard to ensure on other `STM` implementations.

In concurrent programming, it is crucial to be able to *block* when we need to wait for some condition to be true. We can achieve this behavior in STM Haskell using the `retry` function. When `retry` is called, the `atomically` block will immediately terminate. It makes the transaction restart from scratch, with any previous modifications unperformed. The Haskell implementation is also smart enough not to restart the transaction immediately. It will do so only when one or more of the variables involved in the transaction changes. This makes it possible to block the execution of a thread on arbitrary conditions.

Having *blocking* semantics also enables the creation other types, such as `TMVar`. As the name implies, it works as a transactional variant of `MVar`. It blocks when the `TMVar` is full and we try to put a new value on it, or when the `TMVar` is empty and we try to take a value from it. However, it blocks using `retry`, which aborts the transaction it is in and makes it restart

Code 2.10: The `STM` interface

```haskell
-- The STM monad
data STM a
instance Monad STM -- support "do" notation and sequencing

-- Transactional variable
data TVar a
newTVar          :: a -> STM (TVar a)
readTVar         :: TVar a -> STM a
writeTVar        :: TVar a -> a -> STM ()

-- Transactional MVar
data TMVar a
newTMVar         :: a -> STM (TMVar a)
takeTMVar        :: TMVar a -> STM a
putTMVar         :: TMVar a -> a -> STM ()
tryTakeTMVar     :: TMVar a -> STM (Maybe a)
tryPutTMVar      :: TMVar a -> a -> STM Bool

-- Running STM computations
atomically       :: STM a -> IO a
retry            :: STM a
orElse           :: STM a -> STM a -> STM a
```

from the beginning. This makes the order in which the threads are woken up by the scheduler to differ between `MVar` and `TMVar`. `MVar` functions are guaranteed to be woken up one at a time while `TMVar` functions follow the retry semantics. So, while `MVar`s are guaranteed to be fair, `TMVar`s have no such guarantees.

As we can see, Haskell has a robust set of constructs and primitives for concurrent programming. There are several ways we can write an application to achieve a specific goal. Throughout this work, we will study how the choices we make for expressing concurrency in our Haskell programs impacts performance.

3
Measuring Energy Consumption

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.
—EDSGER DIJKSTRA (How do we tell truths that might hurt?)

How to measure the energy consumption of a computer system is a hot topic that expands over a broad area of research. There are several ways we can accomplish this task. They can be categorized into two separate approaches: power measurement and energy estimation. The first one, power measurement, makes use of special power measurement hardware to collect power samples of the running system. These samples are often measured in watts. To obtain the energy (in joules), we have to multiply the power by the time: $E = P \times t$. The second approach, energy estimation, uses software-based techniques to predict how much energy the system is consuming at runtime. It collects data from the running system to be used as predictors of energy consumption. For instance, powertop[1] is a Linux tool that uses this approach. It monitors CPU states, devices drivers and kernel options to report how the active components of the system are behaving regarding power consumption.

For this work, we chose to use an energy estimation approach for measuring the energy consumption of Haskell programs. In Section 3.1, we present more details about RAPL, which is the interface we use. Later, in Section 3.2 and Section 3.3, we present two different performance analysis tools of the Haskell ecosystem: the GHC profiler and Criterion. We explain how these tools work and how we extended them also to analyze energy consumption.

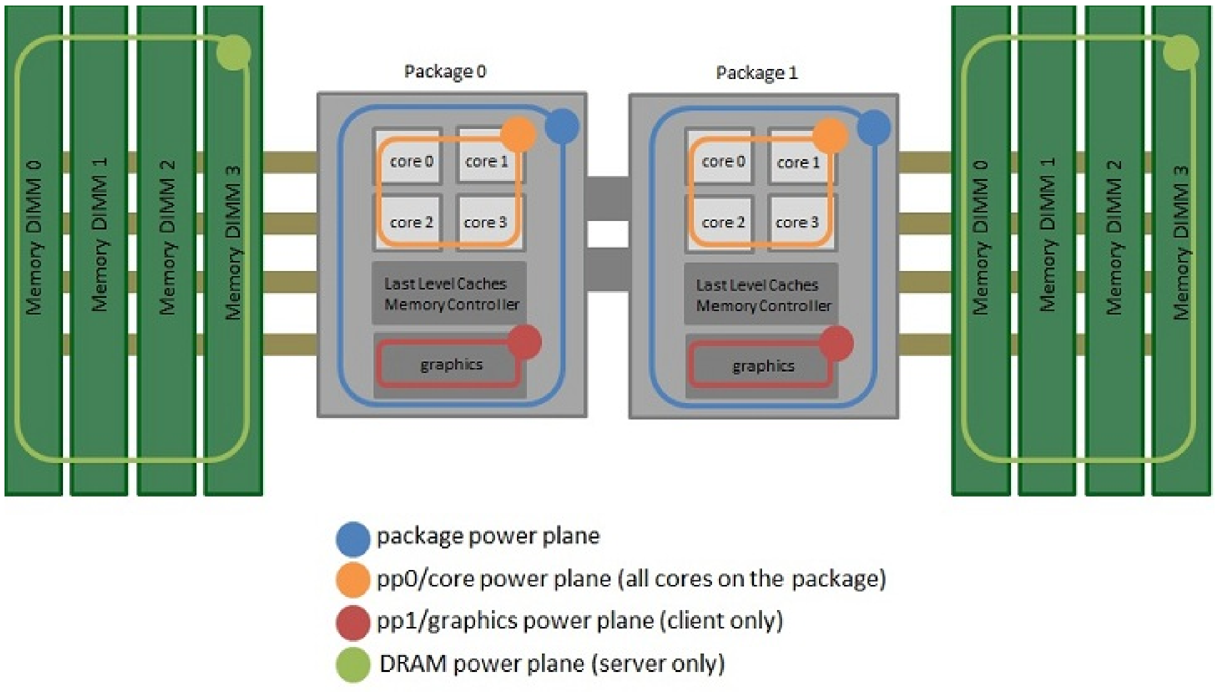
3.1 RAPL

Running Average Power Limit (RAPL) (DAVID et al., 2010) is an interface designed by Intel to enable chip-level power management. It was introduced with the Sandy Bridge microarchitecture. Nowadays, RAPL is widely supported by the Intel architectures, including Xeon family CPUs, that targets server systems, and the popular Core i5 and i7 families, that targets domestic use. This interface provides a set of counters with energy and power con-

[1] https://01.org/powertop

sumption information. To estimate energy consumption, RAPL uses a software power model. This model is based on various hardware performance counters, temperature, leakage models, and I/O models (WEAVER et al., 2012). Its precision and reliability have been extensively studied (ROTEM et al., 2012; HäHNEL et al., 2012).

Figure 3.1: Physical representation of the RAPL domains



Source: "Intel® Power Governor" article (DIMITROV et al., 2012)

With RAPL, developers can monitor energy consumption and set power limits. The access to this information is divided into different domains. Each domain is a physically meaningful domain for power management, as we can see in Figure 3.1. The RAPL domains that are available in a platform vary across product segments. Typically, the desktop platforms have access to {PKG, PP0, PP1}, while the server platforms have access to {PKG, PP0, DRAM}. Each of these domains provides fine-grained reports and control for power management. In Table 3.1, we show what each domain supports. For this work, we are interested in the ENERGY_STATUS information as it provides the current measured energy consumption of a specific domain.

Table 3.1: List of controls supported by the RAPL domains

POWER_UNIT	Provides the scaling factors for each unit
POWER_LIMIT	Allows software to set power limits
ENERGY_STATUS	Reports measured actual energy usage
PERF_STATUS	Reports the performance impact of power limiting
POWER_INFO	Reports power range information for RAPL usage

The interaction with RAPL is done via Machine-Specific Registers (MSRs). Each control listed in Table 3.1 for each domain is a separate MSR. MSRs are special control registers present in the x86 instruction set that are typically used for debugging, monitoring performance, and toggling CPU features. Accessing MSRs requires ring-0 access to the hardware, which is usually only allowed to the operating system kernel. This means that accessing the RAPL readings requires a kernel driver. In Linux, we do not have a specific driver to access the RAPL MSRs. Instead, we have a generic `msr` driver (or kernel module) that exports MSR access to the userspace. The register readings are exposed as files inside the CPU device directories (e.g. `/dev/cpu/0/msr`). These files have read-only permission for superusers (root).

Manipulating these registers is not a very straightforward process. To do this, a developer needs knowledge of system programming and familiarity with the processor instruction set to know how to interpret the raw values exposed by the readings. Also, developers using RAPL need to handle possible register overflows. In a high power consumption scenario, for example, the `ENERGY_STATUS` MSR of the `PKG` domain have a wraparound time of around 60 secs (INTEL, 2016, p. 2465). So to abstract these low-level interfaces from Haskell developers, we present in the next sections two performance analysis tool that we extended to collect energy consumption information using RAPL.

3.2 GHC Profiler

A profiler is a tool for helping the development of efficient programs. Its main function is to provide the necessary information for developers to identify performance bottlenecks. Once the hot spots in a program have been identified, the developer can work on the code to improve its performance and continuously check the effect of each modification. To achieve this goal, a profiler should keep track of important program resources. Moreover, the data gathered by the profiler must be related to the program source code in a way that is meaningful to the developer. This is usually accomplished by reporting the measurements by program structures (e.g. functions or methods) or source code structures (e.g. lines).

However, it is hard to establish this correspondence between measurements and source code for high-level languages. Usually, these languages provide abstractions and constructs that are unrelated to the way that the underlying execution engine works. Haskell is not different. Features such as polymorphism, high-order functions, and lazy evaluation make this task even harder. For example, in the presence of lazy evaluation, the evaluation of an expression can be interleaved with the evaluation of the inputs that this expression demands. This makes the resulting order of execution of expressions bear no resemblance to the source code.

To address this problem, the GHC profiler uses *cost centres* (SANSOM; PEYTON-JONES, 1995). They are the logical components of the program to which the profiler associates the gathered data. A cost centre is simply a label to which we attribute execution costs. They are represented by annotations around expressions. So the costs incurred by the evaluation of the annotated expression are assigned to the enclosing cost centre. The cost centres can be automatically generated by the compiler or manually specified by the developer through the `{-#`

SCC #-} directive. In Code 3.1[2], we show an example of how it can be manually specified by the developer (line 10). In this example, all the costs incurred by the evaluation of `sum xs / fromIntegral (length xs)` will be attributed to the `mean` cost centre. It is important to point out that cost centres have a formally defined semantics (SANSOM; PEYTON-JONES, 1995) that specify how the cost attribution works.

Code 3.1: Haskell program to calculate the mean of a list of numbers

```
1  import System.Environment
2  import Text.Printf
3
4  main :: IO ()
5  main = do
6      [d] <- map read `fmap` getArgs
7      printf "%f\n" (mean [1..d])
8
9  mean :: [Double] -> Double
10 mean xs = {-# SCC mean #-} sum xs / fromIntegral (length xs)
```

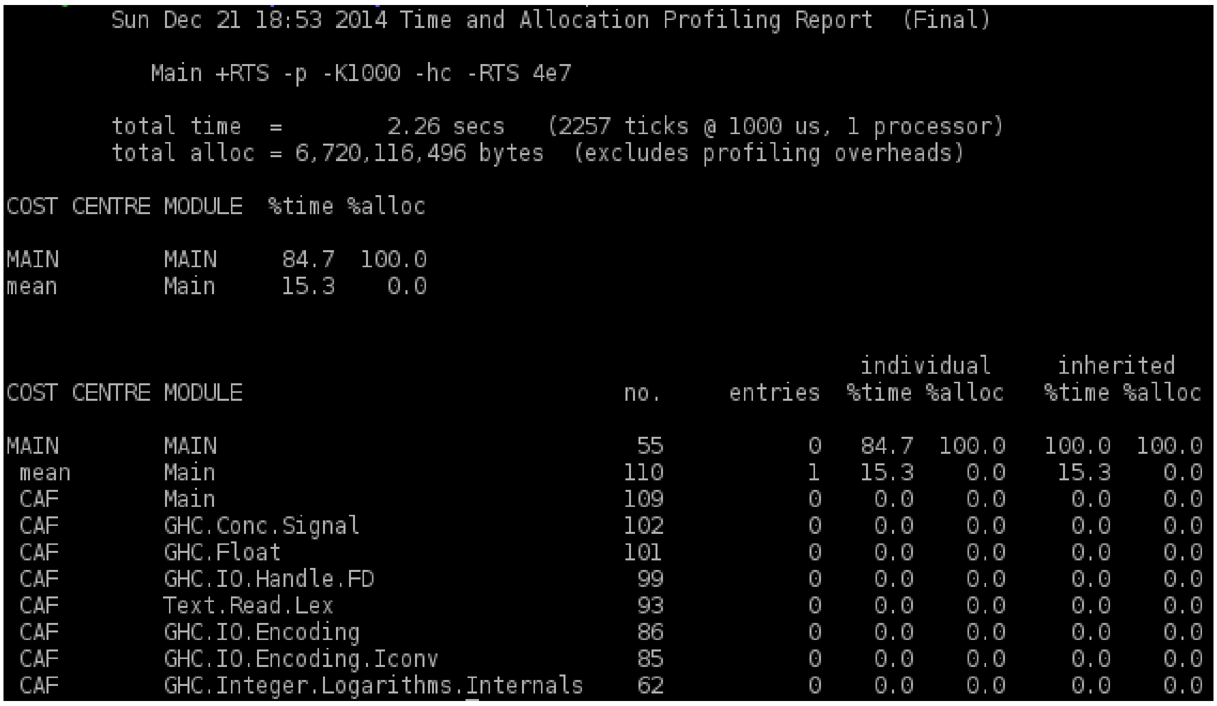
Currently, the GHC profiler is capable of measuring time and space usage. In GHC, the profiler is part of the runtime system, which means that the profiling routines are contained within the final program executable. So to build a program for profiling, we need to pass to GHC the `-prof` flag when compiling it. Then, to run this program in profiling mode, we need to specify it to the runtime system by passing the `+RTS -p` argument. It makes the runtime system collect time and memory usage data from the execution to produce a detailed report at the end. Figure 3.2 shows a profiling report for the program in Code 3.1.

As we can see in this example, we can split the report into three different sections. The first one is the header. It shows how the program was executed (which flags and arguments were passed to run it), and the total time and memory allocated during the whole execution of the program. The second part shows a break-down of the most costly cost centres. In this case, the top-level `MAIN` and the user-defined `mean`.

The third section shows a break-down by *cost centre stack* (MORGAN; JARVIS, 1998). A cost centre stack is similar to a call graph. It defines a hierarchy of cost centres. In this case, we can see that `MAIN` is the root of the cost centre stack, followed by `mean` and some instances of `CAF` as children. A Constant Applicative Form (CAF) is an expression that contains no free variables, i.e. it is a constant expression. So a `CAF` cost centre represents the one-off costs of evaluating such constants. Also, in this section of the report, we have multiple columns showing the profiling data. The time and memory usage percentage columns are shown into two different groups: `individual` and `inherited`. The former is the total of program resources spent by this cost centre while the latter is the total of program resources spent by this cost centre and its

[2]This code example was extracted from (O'SULLIVAN; GOERZEN; STEWART, 2008)

Figure 3.2: Example of profiling report for the `mean` program



```
        Sun Dec 21 18:53 2014 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K1000 -hc -RTS 4e7

        total time  =        2.26 secs   (2257 ticks @ 1000 us, 1 processor)
        total alloc = 6,720,116,496 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

MAIN        MAIN     84.7  100.0
mean        Main     15.3    0.0


                                                      individual     inherited
COST CENTRE MODULE                        no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                           55           0   84.7  100.0   100.0  100.0
 mean       Main                          110           1   15.3    0.0    15.3    0.0
 CAF        Main                          109           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal               102           0    0.0    0.0     0.0    0.0
 CAF        GHC.Float                     101           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Handle.FD               99           0    0.0    0.0     0.0    0.0
 CAF        Text.Read.Lex                  93           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding                86           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv          85           0    0.0    0.0     0.0    0.0
 CAF        GHC.Integer.Logarithms.Internals 62         0    0.0    0.0     0.0    0.0
```

Source: Generated by the GHC profiler

children. The other columns are `no.`, which is the id of the cost centre, and `entries`, which is the number of times that the expression enclosed by this cost centre was evaluated.

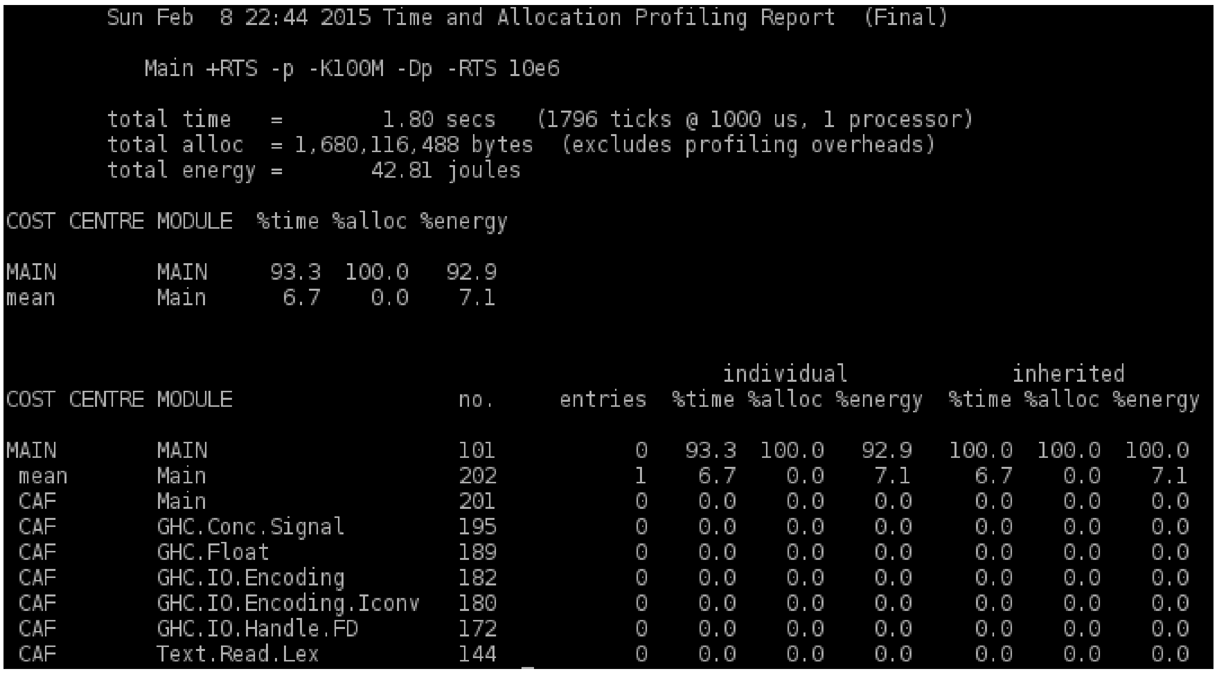
So to help on our journey of understanding the energy behavior of Haskell programs, we have extended the GHC profiler to add a new metric: energy consumption. The idea is to collect energy consumption readings from RAPL and use the GHC infrastructure to calculate the energy spent by each cost centre. Having this, we can display two new columns in the final report accounting the percentage of energy consumed by each cost centre. We based our solution on the approach used by the time profiler. To measure the execution time, the profiler keeps in each cost centre a tick counter. At any moment, the cost centre that is currently executing is held in a special register by the runtime system. Then, a regular clock interrupt[3] runs a routine to increment this tick counter of the cost centre in execution. At the end of the program execution, the value held in the tick counter of each cost centre enables the profiler to determine and report the relative execution time cost of the different parts of the program.

Similarly, our extended version of the GHC profiler keeps in each cost centre an accumulator. At each clock interrupt, the profiler adds to the accumulator of the cost centre that is currently in execution the energy consumed between the previous and current interrupt. This is accomplished by always saving the previous and current energy readings obtained from RAPL. If an overflow occurs on RAPL during the (extremely small) interval between two consecutive interrupts, we do not update the accumulator in this tick. At the end of the program execution, the

[3]By default this interval is 20ms. However, the user can change it by passing a custom value to the runtime system via the `-V` argument.

profiler will be able to report the energy consumed by each cost centre based on its accumulator value. Figure 3.3 shows the report of our extended GHC profiler the same program from Code 3.1. As we can see in this report, we have now energy consumption information on each section of the report. In the header, we have the total energy consumed during the execution of the program. In the second and third sections, we have extra columns showing the percentage of energy consumed by each cost centres.

Figure 3.3: Example of profiling report with energy metrics for the `mean` program



```
        Sun Feb  8 22:44 2015 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K100M -Dp -RTS 10e6

        total time   =        1.80 secs   (1796 ticks @ 1000 us, 1 processor)
        total alloc  = 1,680,116,488 bytes  (excludes profiling overheads)
        total energy =       42.81 joules

COST CENTRE MODULE  %time %alloc %energy

MAIN        MAIN     93.3  100.0   92.9
mean        Main      6.7    0.0    7.1


                                                      individual             inherited
COST CENTRE MODULE                 no.     entries  %time %alloc %energy  %time %alloc %energy

MAIN        MAIN                   101           0   93.3  100.0    92.9  100.0  100.0  100.0
 mean       Main                   202           1    6.7    0.0     7.1    6.7    0.0    7.1
 CAF        Main                   201           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        GHC.Conc.Signal        195           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        GHC.Float              189           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        GHC.IO.Encoding        182           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        GHC.IO.Encoding.Iconv  180           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        GHC.IO.Handle.FD       172           0    0.0    0.0     0.0    0.0    0.0    0.0
 CAF        Text.Read.Lex          144           0    0.0    0.0     0.0    0.0    0.0    0.0
```

Source: Generated by the GHC profiler

Our modified version of GHC is publicly available on GitHub[4]. It is based on GHC 7.10 and includes the profiler with energy metrics. However, it is important to note that our implementation has one limitation. We can provide more accurate results only for programs that use a single capability. The main reason for this is the fact that RAPL does not provide the energy consumed by a single core. It provides only the energy consumption of the cores altogether, so if two cost centres are executing in parallel, we cannot tell how much energy each one consumed separately. However, we could easily adapt the current implementation to handle this case once the underlying API provides the energy by core information. Also, it is easy enough to change the use of RAPL for another API that provides energy consumption information if needed.

Despite this limitation, the profiler tool is particularly useful for providing fine-grained information about energy consumption. It enables developers to find the energy hot spots of Haskell programs. In the next section, we present another performance analysis tool called Criterion. Different from a profiler, it uses a coarse-grained approach for evaluating the performance of a program.

[4]http://github.com/green-haskell/ghc/tree/wip/green

3.3 Criterion

Criterion (O'SULLIVAN, 2009a) is a microbenchmarking library that is used to measure the performance of Haskell code. Its main objective is to estimate the cost of running a small, independent piece of code. At a first glance, it may seem that it does the same job of a profiler, but this is not the case. Criterion is not designed to find hot spots in a program. Instead, it is useful for analyzing the cost of a given operation. A profiler is about taking a snapshot of a single execution of a program and reporting fine-grained information about its performance. Criterion, however, is about running a certain piece of code several times to analyze its performance. It reports to the developer a statistically-backed estimation of the cost of running the selected piece of code once. As Criterion does not analyze the performance for each cost centre, only the benchmarked code as a whole, we say its analysis is coarse-grained.

Code 3.2: Definition of a Criterion benchmark for the `fib` function

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0      = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

Criterion provides a framework for both executing benchmarks as well as analyzing their results. This framework is based on a simple API that hides most of the complexity involved in performing benchmarks. In Code 3.2[5], we show an example of how to define a Criterion benchmark. Here, we want to analyze the performance of the `fib` function. The benchmark is defined to execute `fib` passing nine as argument. In this example, we are using three important functions of Criterion's API:

- `defaultMain`: takes care of executing a set of benchmarks
- `bench`: creates a benchmark based on an action provided by the developer
- `whnf`: makes sure the benchmarked action is evaluated to weak head normal form to stop it from being evaluated only once due to Haskell's laziness

[5]This example was extracted from <http://www.serpentine.com/criterion/tutorial.html>

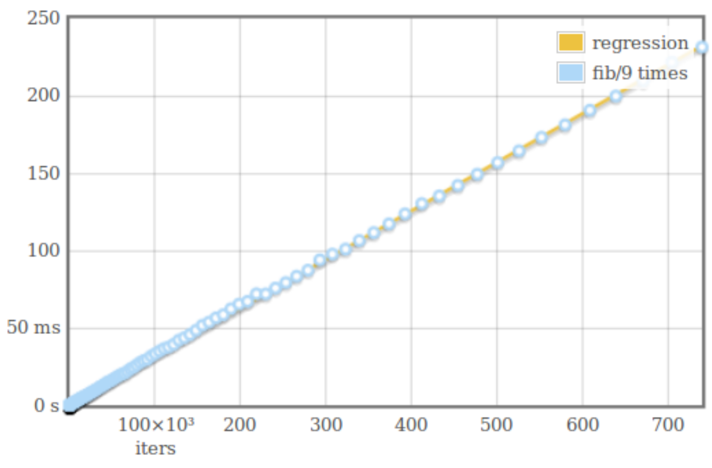
Figure 3.4: Output of the `fib` benchmark

```
benchmarking fib/9
time                 314.4 ns   (312.2 ns .. 318.5 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 315.3 ns   (314.0 ns .. 319.4 ns)
std dev              7.081 ns   (1.625 ns .. 14.63 ns)
variance introduced by outliers: 26% (moderately inflated)
```

Source: Generated by Criterion

In Figure 3.4, we show the output for the benchmark described in Code 3.2. The first line shows `time`, which is the estimation of the time needed for executing `fib 9` once. It is obtained using an Ordinary Least Squares (OLS) regression model. Inside the parenthesis, after the time estimation, we can see the lower and upper bounds of the confidence interval that is calculated using *bootstrapping* (DAVISON; HINKLEY, 1997). It means that when randomly resampling the data, 95% of estimates fell between the lower and upper bounds of the interval. So the quality of the estimation is better when the bounds are closer to its value. The second line shows the coefficient of determination, or R^2, which is a statistical measure of how well the linear regression model approximates the observed measurements. An R^2 between 0.99 and 1 indicates an excellent approximation. The `mean` and `std dev` lines are the mean execution time and the standard deviation, respectively. Finally, the last line indicates the degree to which the standard deviation is inflated by outlying measurements.

Figure 3.5: Measurement chart generated for the `fib` benchmark



Source: Generated by Criterion

This kind of information that Criterion provides is quite useful for making sure that the measurements are not being tainted by external factors such as other loads on the operating system. Alongside with the textual report, Criterion can also generate some charts like the one shown in Figure 3.5. In this graph, the *x*-axis indicates the number of loop iterations, while the *y*-axis shows the execution time for a given number of iterations. The blue circles are the raw measurements while the orange line is the linear regression generated from this data. The closer the dots are from the line; the better is the regression model.

As we can see from this example, each sample that Criterion uses for the regression model corresponds to the measurements collected during a distinct number of consecutive executions of the benchmarked code. This specific number of loop iterations is the independent variable of the regression model, which the author calls `iters`. The number of different `iters` that Criterion performs for a given benchmark depends on how much time the benchmarked code takes to run. Criterion tries to run it as much as possible so that: (1) it collects enough measurements for properly resampling the data, and (2) it generates enough measurements that have long spans of execution to outweigh the cost of measurement. For short-lived benchmarks such as Code 3.2, in the order of nanoseconds, we can see that Criterion collects several samples with a high number of consecutive iterations (in the order of a hundred thousand). For longer benchmarks, in the order of tens of seconds, it is guaranteed that Criterion will collect at least five samples, from one to five iterations each, respectively[6].

Figure 3.6: Output with CPU cycles of the `fib` benchmark

```
benchmarking fib/9
time                 317.2 ns   (314.2 ns .. 319.4 ns)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 314.4 ns   (313.3 ns .. 315.8 ns)
std dev              4.117 ns   (2.682 ns .. 5.398 ns)
cycles:              0.999 R²   (0.999 R² .. 1.000 R²)
  iters              1079.434   (1069.292 .. 1087.144)
  y                  924904.370 (562772.048 .. 1358678.998)
variance introduced by outliers: 13% (moderately inflated)
```

Source: Generated by Criterion

Besides elapsed time, Criterion can also perform linear regression on other metrics such as CPU time, CPU cycles, bytes allocated, and the number of garbage collections. It can be done by passing the other metric as a command line argument when running the benchmark. For example, passing `--regress cycles:iters` will regress the number of CPU cycles against the number of loop iterations. Figure 3.6 shows the output for such example for Code 3.2. As we can see, there are three new lines in the report. The first one corresponds to the R^2 for this new regression. The second line we have the estimation of how much cycles each iteration costs, which is the slope of the regression curve. The last line is where the curve intercepts the *y*-axis. This feature allows for developers to have a good overview from different perspectives about the performance of the benchmarked code.

To take advantage of Criterion in our work, we have extended it to add a new metric:

[6]This information is not available in the library documentation. We got it by inspecting the source code.

energy consumption. The fact that Criterion's infrastructure already supports estimating the cost of other metrics than elapsed time helped us to modify it in a "cleaner" way. However, the way the energy data is collected is different from the other metrics because RAPL registers are susceptible to overflows. In Code 3.3, we show how Criterion currently collects the measurements. As we can see, it saves the counters before and after it executes the benchmarked code `iters` times. If we used the same approach for energy consumption, in the presence of an overflow, the data collected would be inconsistent.

Code 3.3: Internal function that execute the benchmarks in Criterion

```haskell
measure :: Benchmarkable -> Int64 -> IO (Measured, Double)
measure (Benchmarkable run) iters = do
  startStats <- getGCStats
  startTime <- getTime
  startCpuTime <- getCPUTime
  startCycles <- getCycles
  run iters
  endTime <- getTime
  endCpuTime <- getCPUTime
  endCycles <- getCycles
  endStats <- getGCStats
  let !m = applyGCStats endStats startStats $ measured {
             measTime     = max 0 (endTime - startTime)
           , measCpuTime = max 0 (endCpuTime - startCpuTime)
           , measCycles  = max 0 (fromIntegral (endCycles - startCycles))
           , measIters   = iters }
  return $ (m, endTime)
```

To overcome this problem, we collect the energy consumption differently from the other metrics. We have to read the RAPL counters constantly during the benchmark execution in order to detect overflows. One way we could accomplish this task is by spawning a background thread to collect the energy data. However, this approach is not very lightweight. It would increase the measurement overhead and possibly influence the result of more sensitive benchmarks. So instead, we have used native Linux signals. It works as follows: before Criterion runs the benchmarked code, we register a Linux signal to be fired every 10ms. When registering this signal, we attach a signal handler that we implemented. Inside this handler, we read the RAPL counters, check if an overflow happened and, if not, we save the energy consumed between the last two readings into an internal accumulator. At the end, when the benchmark finishes executing, we can deregister the signal and get the value held by the accumulator, which is the energy measurements.

To implement this solution, similarly to how the time and cycles data is collected, we use native C code. The communication with the Criterion code, which is implemented in Haskell, is done straightforwardly via foreign function interface (FFI) calls. The interface with Linux to use signals is done using the `sigaction` and `setitimer` syscalls. We developed this extension with the cooperation of our colleagues Gilberto Melfe and João Paulo Fernandes from University of Beira Interior. The source code of our modified version of Criterion is publicly available on GitHub[7]. In Figure 3.7, we show an example of a benchmark output using Criterion with the energy metrics. In this case, the benchmark was executed passing the `--regress energy:iters` argument to regress energy consumption against the number of loop iterations. As we can see in the output, a single execution of this program is expected to consume 180.9 joules of energy.

Figure 3.7: Output example of Criterion with energy metrics

```
benchmarking dining-philosophers (forkOS | MVar)
time                 2.183 s    (1.915 s .. 2.510 s)
                     0.997 R²   (0.991 R² .. 1.000 R²)
mean                 2.179 s    (2.113 s .. 2.212 s)
std dev              57.17 ms   (0.0 s .. 57.19 ms)
energy:              0.999 R²   (0.997 R² .. 1.000 R²)
  iters              180.947    (164.629 .. 200.826)
  y                  0.937      (-71.359 .. 37.230)
variance introduced by outliers: 19% (moderately inflated)
```

Source: Generated by Criterion

Criterion is a powerful tool. It provides a robust methodology to benchmark Haskell code. In the next chapter, we present an empirical study where Criterion was heavily employed to measure the performance and energy consumption of various concurrent Haskell programs.

[7]<http://github.com/green-haskell/criterion>

4

The Energy Efficiency of Concurrent Haskell

In this chapter, we present an empirical study evaluating the performance and energy consumption characteristics of three thread management constructs and three data-sharing primitives of Concurrent Haskell. We start by providing a brief overview (Section 4.1) of the study and state the research questions. Section 4.2 describes the benchmark we developed, the environment and the methodology we used. Sections 4.3, 4.4, and 4.5 present our results. Finally, Section 4.6 list the threats to validity.

4.1 Overview

Recently, the software engineering community has been showing a keen interest in researching about the development of energy-efficient software. As multicore architectures are the norm nowadays, this interest also extends for energy-efficiency from the perspective of concurrent software running on multicore architectures (TREFETHEN; THIYAGALINGAM, 2013; RIBIC; LIU, 2014; PINTO; CASTOR; LIU, 2014b). However, the same cannot be said about functional programming. This is unfortunate as functional programming languages are seen as a good alternative for making concurrent programming less error-prone. Particularly, Haskell provides a solid foundation for building concurrent software, and we know very little about its energy behavior.

We believe that the first step towards optimizing energy consumption of concurrent programs is to gain a comprehensive understanding of their energy behaviors. This chapter presents an empirical study that aims to understand the energy behaviors of Haskell concurrent programs on multicore architectures. In particular, our study focuses on how programmer's decisions may impact energy consumption and performance. The following questions motivate our research:

RQ1. Do alternative *thread management constructs* have different impacts on energy consumption?

RQ2. Do alternative *data-sharing primitives* have different impacts on energy consumption?

RQ3. What is the relationship between the *number of capabilities* and energy consumption?

To answer **RQ1**, we select three thread management constructs: `forkIO`, `forkOn` and `forkOS`. As we saw in Section 2.3, these are the three functions available in Haskell to spawn a new thread of execution. To answer **RQ2**, we select three data-sharing primitives: `MVar`, `TMVar` and `TVar`. Given these constructs, our investigation aims to understand how the energy consumption can be impacted by changing the concurrency primitives used in a Haskell program. Finally, to answer **RQ3**, we explore various capabilities settings to see how it can impact energy consumption and performance on a multicore environment.

4.2 Study Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

4.2.1 Benchmarks

We selected a variety of concurrent Haskell programs to use as benchmarks in our study, listed as follows. Benchmarks 1-6 are from The Computer Language Benchmarks Game (CLBG)[1]. CLBG is a benchmark suite aiming to compare the performance of various programming languages. Benchmark 7 is from Rosetta Code [2], a code repository of solutions to common programming tasks. Benchmarks 8-9 were developed by us.

1. `chameneos-redux`: In this benchmark chameneos creatures go to a meeting place and exchange colors with a meeting partner. It encodes symmetrical cooperation between threads.

2. `fasta`: This benchmark generates random DNA sequences and writes them in FASTA format[3]. The size of the generated DNA sequences is in the order of hundreds of megabytes. In this benchmark, each worker synchronizes with the previous one to output the sub-sequences of DNA in the correct order.

3. `k-nucleotide`: This benchmark takes a DNA sequence and counts the occurrences and the frequency of nucleotide patterns. This benchmark employs string manipulation and hashtable updates intensively. There is no synchronization in the program besides the main thread waiting for the result of each worker.

4. `mandelbrot`: A mandelbrot is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape. Mandelbrot set images are created by sampling complex numbers and determining for each one whether the result tends toward infinity when a particular mathematical operation is

[1]http://benchmarksgame.alioth.debian.org/
[2]http://rosettacode.org/
[3]https://en.wikipedia.org/wiki/FASTA_format

iterated on it. The only synchronization point in this benchmark is the main thread waiting for the result of each worker.

5. `regex-dna`: This benchmark implements a string-based algorithm that performs multiple regular expression operations, match and replace, over a DNA sequence. The only synchronization point is the main thread waiting for the result of each worker.

6. `spectral-norm`: The spectral norm is the maximum singular value of a matrix. It synchronizes the workers using a cyclic barrier.

7. `dining-philosophers`: An implementation of the classical concurrent programming problem posed by Dijkstra (1971). The philosophers perform no work besides manipulating the forks and printing a message when eating.

8. `tsearch`: A parallel text search engine. This benchmark searches for occurrences of a sentence in all text files in a directory and its sub-directories. It is based on a previous empirical study comparing STM and locks (PANKRATIUS; ADL-TABATABAI, 2011).

9. `warp`: Runs a set of queries against a Warp server retrieving the resulting webpages. Warp is the default web server used by the Haskell Web Application Interface (WAI), part of the Yesod Web Framework. This benchmark was inspired by the Tomcat benchmark from DaCapo (BLACKBURN et al., 2006).

We selected the benchmarks based on their diversity. For instance, `chameneos-redux` and `dining-philosophers` are synchronization-intensive programs. `mandelbrot` and `spectral-norm` are CPU-intensive and scale well on a multicore machine. `k-nucleotide` and `regex-dna` are CPU- and memory-intensive, while `warp` is IO-intensive. `tsearch` combines IO and CPU operations, though much of the work it performs is CPU-intensive. `fasta` is peculiar in that is CPU-, memory-, synchronization- and IO-intensive.

Also, some benchmarks have a fixed number of workers (`chameneos-redux`, `k-nucleotide`, `regex-dna`, and `dining-philosophers`) and others spawn as many workers as the number of capabilities (`fasta`, `mandelbrot`, `spectral-norm`, `tsearch` and `warp`). For the `dining-philosophers` benchmark, it is possible to establish prior to execution the number of workers.

4.2.2 Experimental Environment

For our study, all experiments were conducted on a machine with 2x10-core Intel Xeon E5-2660 v2 processors (Ivy Bridge microarchitecture), 2.20 GHz, with 256GB of DDR3 1600MHz memory. It has three cache levels (L1, L2, L3): L1 with 32KB per core, L2 with

256KB per core, and L3 with 25MB per socket (Smart cache). This machine runs the Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25) operating system, GHC 7.10.2, and our modified Criterion based on version 1.1.0 of the official release (Section 3.3).

All experiments were performed with no other load on the OS. We conform to the default settings of the operating system. For each benchmark, we used the same compilation and runtime flags employed in the original benchmark suites. This is important to preserve the same performance characteristics as intended by the implementers. Both the energy consumption and the performance are measured by Criterion.

4.2.3 Methodology

Most of the benchmarks we used were implemented in their respective suites using a single thread management construct and a single data-sharing primitive. For example, the ones from CLBG use originally `forkIO` and `MVar`. In order to analyze the impact of both thread management constructs and data-sharing primitives, we manually refactored each benchmark to create new *variants* using different constructs. Changing the thread management construct is a straightforward process. The functions have almost the same signature. The only exception is `forkOn` that requires an extra argument representing in which capability the thread will run. For this study, we distributed the threads evenly among the capabilities when using `forkOn`. However, changing the data-sharing primitive is not always simple since the semantics of `TVars` and `MVars` differ considerably. To properly perform this refactoring, we used the techniques proposed by Soares-Neto (2014) to refactor the benchmarks to use STM.

As a result of this process, each benchmark has up to nine distinct variants covering a number of different combinations of both thread management constructs and data-sharing primitives. However, it is important to note that there are some cases like `dining-philosophers` where not all possible combinations were created. In this particular implementation, the shared variable is also used as a condition-based synchronization mechanism. In such cases, where `MVars` are used to implement both mutual exclusion and condition-based synchronization, we did not create the `TVar` variants. We limited ourselves to the variants using `MVars` and `TMVars` since a `TMVar` is very similar to an `MVar` while executing inside a transaction. In other benchmarks like `tsearch` and `warp`, we changed only the thread management construct as they are complex applications and it would not be straightforward to change the synchronization primitives without introducing potential bugs.

In the end, each variant we created is represented by a standalone executable. This executable is a Criterion microbenchmark that performs the experiment by calling the original program entry point multiple times. We run each benchmark under nine different configurations of capabilities. We used the following values for `N`: $\{1,2,4,8,16,20,32,40,64\}$. Where 20 and 40 are the number of physical and virtual cores, respectively.

In Appendix A, we provide an example of one of the benchmarks we used. We show the description of the problem that this benchmark solves and the implementation of each variant we generated to use in this study. It can also be accessed on GitHub at <http://github.com/green-haskell/concurrency-benchmark>. In this repository, we made publicly available the implementation of all benchmarks variants as well as the scripts and instructions necessary to replicate this experiment.

4.3 Study Results

In this section, we report the results of our experiments with concurrent Haskell programs. The results are presented in Figures 4.1, 4.2 and 4.3. Here, the odd rows are energy consumption results, while the even rows are the corresponding execution time results. We omitted the experiments using 64 capabilities in order to make the charts more readable. The charts including this configuration as well as all the data and source code used in this study are available at green-haskell.github.io.

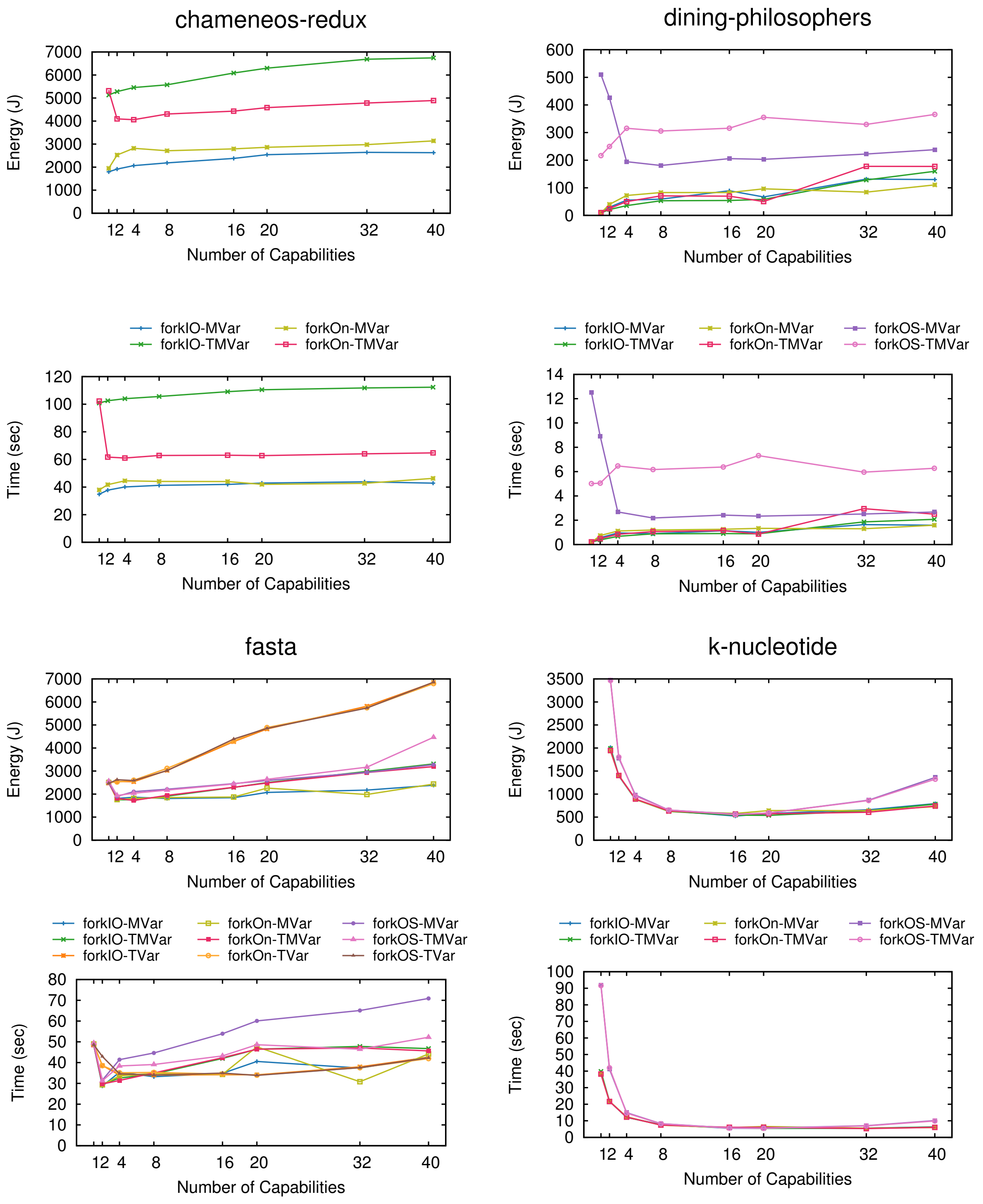
Small changes can produce big savings. One of the main findings of this study is that simple refactorings such as switching between thread management constructs can have considerable impact on energy usage. For example, in `spectral-norm`, using `forkOn` instead of `forkOS` with `TVar` can save between 25 and 57% energy, for a number of capabilities ranging between 2 and 40. Although the savings vary depending on the number of capabilities, for `spectral-norm`, `forkOn` exhibits lower energy usage independently of this number. For `mandelbrot`, variants using `forkOS` and `forkOn` with `MVar` exhibited consistently lower energy consumption than ones using `forkIO`, independently of the number of capabilities. For the `forkOS` variants, the savings ranged from 5.7 to 15.4% whereas for `forkOn` variants the savings ranged from 11.2 to 19.6%.

This finding also applies to data sharing primitives. In `chameneos-redux`, switching from `TMVar` to `MVar` with `forkOn` can yield energy savings of up to 61.2%. Moreover, it is advantageous to use `MVar` independently of the number of capabilities. In a similar vein, in `fasta`, going from `TVar` to `MVar` with `forkIO` can produce savings of up to 65.2%. We further discuss the implications of this finding in Section 4.5.

Faster is not always greener. Overall, the shapes of the curves in Figures 4.1, 4.2 and 4.3 are similar. Although, for six of our nine benchmarks, in at least two variants of each one, there are moments where faster execution time leads to a higher energy consumption. For instance, in the `forkOn-TMVar` variant of `regex-dna`, the benchmark is 12% faster when varying the number of capabilities from 4 to 20 capabilities. But at the same time, its energy consumption increases by 51%. Also, changing the number of capabilities from 8 to 16 in the `forkIO` variant of `tsearch` makes it 8% faster and 22% less energy-efficient.

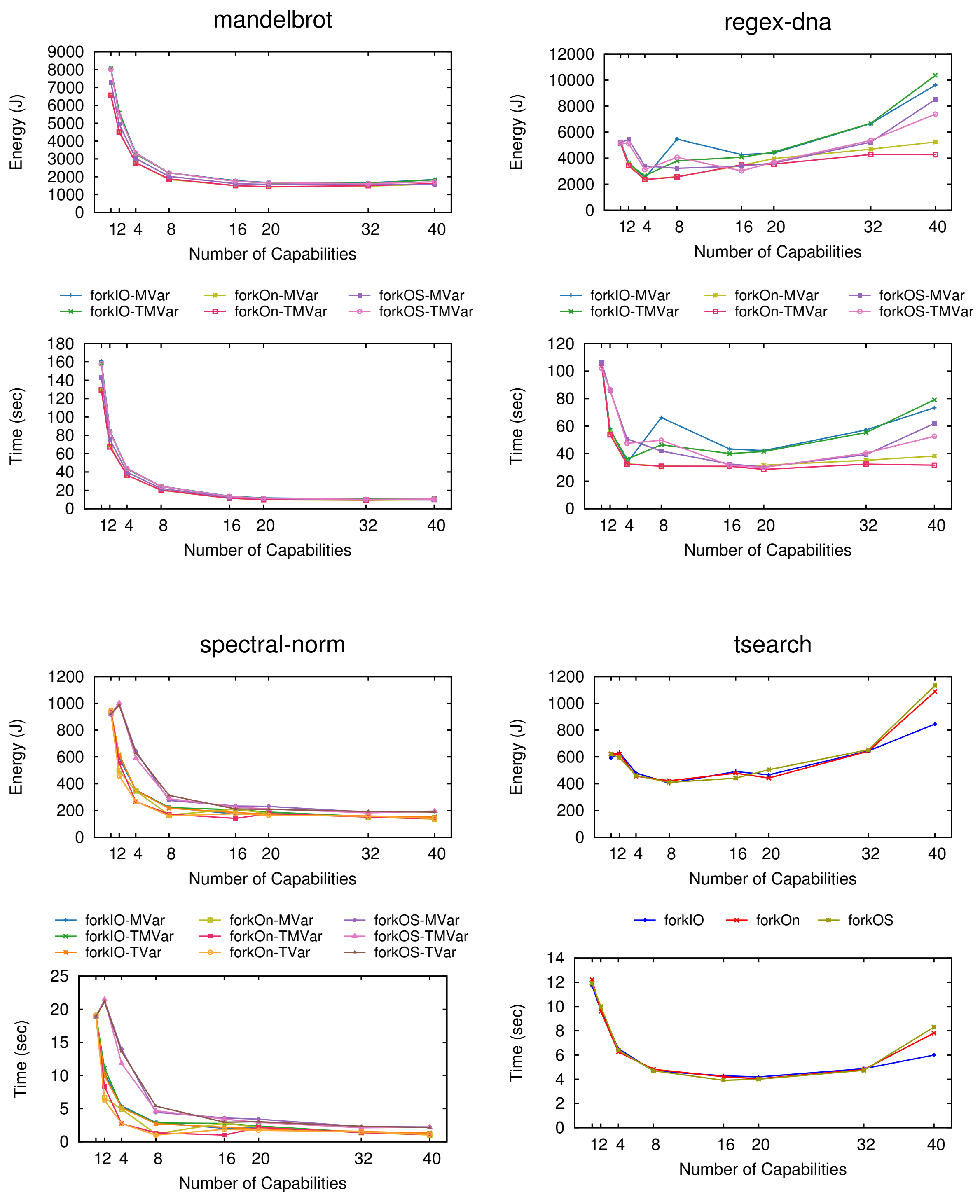
In one particular benchmark, `fasta`, we had strongly divergent results in terms of

Figure 4.1: Energy and Time results for Benchmarks 1, 2, 3 and 7



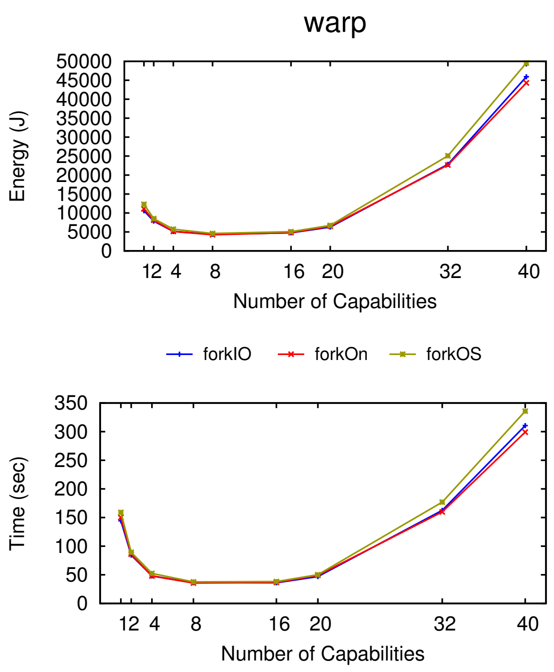
Source: Made by the author

Figure 4.2: Energy and Time results for Benchmarks 4, 5, 6 and 8



Source: Made by the author

Figure 4.3: Energy and Time results for Benchmark 9



Source: Made by the author

performance and energy consumption for some of the variants. For this benchmark, the variants employing `TVar` outperformed the ones using `TMVar` and `MVar`. For example, when using a number of capabilities equal to the number of physical cores of the underlying machine (20), the `forkOS-TVar` variant was 43.7% faster than the `forkOS-MVar` one. At the same time, the `TVar` variants exhibited the worst energy consumption. In the aforementioned configuration, the `forkOS-TVar` variant consumed 87.4% more energy. We analyze this benchmark in details in Section 4.4.

There is no overall winner. Overall, no thread management construct or data sharing primitive, or combination of both is the best. For example, the `forkIO-TMVar` variant is one of most energy-efficient for `dining-philosophers`. The `forkOS-TMVar` variant consumes more than six times more energy. However, for the `chameneos-redux` benchmark, the `forkIO-TMVar` variant consumes 2.4 times more energy than the best variant, `forkIO-MVar`. This example is particularly interesting because these two benchmarks have similar characteristics. Both `dining-philosophers` and `chameneos-redux` are synchronization-intensive benchmarks and both have a fixed number of worker threads. Even in a scenario like this, using the same constructs can lead to discrepant results.

Choosing more capabilities than available CPUs is harmful. The performance of most benchmarks is severely impaired by using more capabilities than the number of available CPUs. In `chameneos-redux`, for example, moving from 40 to 64 capabilities can cause a 13x slowdown. This suggests that the Haskell runtime system was not designed to handle cases where capabilities outnumber CPU cores. We discuss this matter further in Chapter 5.

4.4 Case study: `fasta`

This section expands the analysis of the `fasta` benchmark as it behaves differently from the other benchmarks.

4.4.1 How it works

In `fasta`, each worker thread executes independently, generating a piece of the resulting DNA sequence. This piece is generated using a set of random numbers that are calculated by each worker based on a seed value. The current seed is kept in a shared variable. Each worker takes it, calculates the random numbers, and puts a new seed back. The following steps can describe the workers' loop:

1. Take *seed0* from the shared variable
2. Generate *random_numbers* and *seed1*
3. Put *seed1* on the shared variable
4. Compute the DNA sequence based on *random_numbers*
5. Wait until the predecessor DNA sequence is written to output
6. Write DNA sequence to output

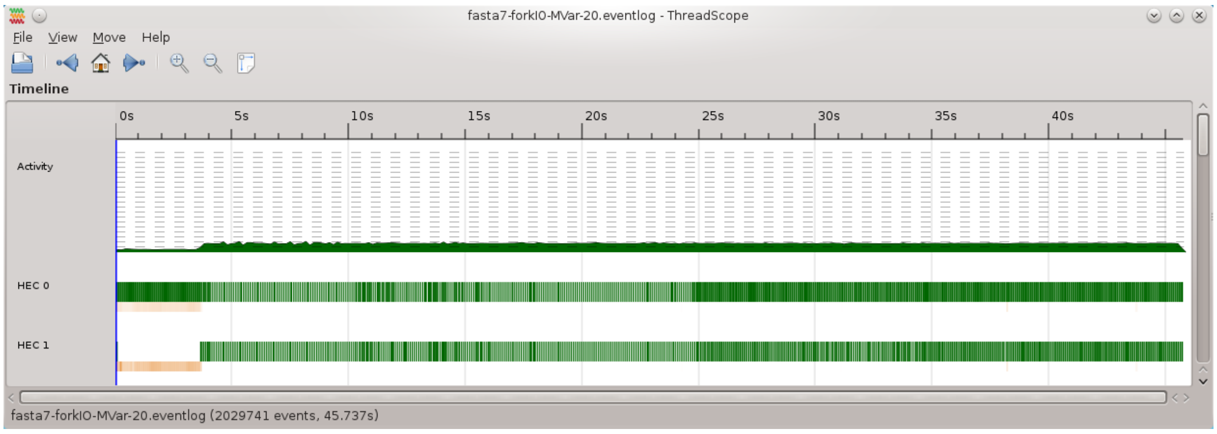
As mentioned above, the worker thread has to wait (if needed) to write its DNA sequence after the one generated using the predecessor seed. This step is necessary to guarantee that the final DNA sequence is assembled in the correct order. However, it takes approximately the same time for each worker to generate its piece of DNA sequence as the sequences have the same size. So this particular blocking behavior is not a bottleneck as it takes more time to compute the DNA sequence than to write it to output.

4.4.2 The fastests consume more energy

In Figure 4.2, looking at the results for the `TVar` variants of `fasta`, we can see they have a peculiar behavior. First, execution time and energy consumption are pretty similar for all three variants and for each capabilities settings. So every thread management strategy impacts performance in the same way for the `TVar` variants. This behavior is unusual in other benchmarks where the combination of both thread management strategy and synchronization primitive seems to affect performance differently. Second, the `TVar` variants have the best overall execution time while they are the least energy-efficient. This is also an unexpected behavior. In the other benchmarks, when comparing the charts for execution time and energy consumption for the same benchmark, the ordering of the curves is preserved for most cases.

In order to better understand this scenario, we have used the `eventlog`[4] feature of GHC to profile the execution of `fasta`. The log records the activity of the Haskell runtime system throughout the whole program execution. The ThreadScope (JONES JR.; MARLOW; SINGH, 2009) readings of the log can be seen in Figures 4.4 and 4.5 for the `forkIO-MVar` and `forkIO-TVar` variants, respectively. The first row of the report shows the overall CPU activity while the others show the activity on each capability (or HEC, as named in the pictures). In this particular example, we executed both variants using 20 capabilities. Although only the first two capabilities are shown in these images, the activity of the other capabilities behaves similarly.

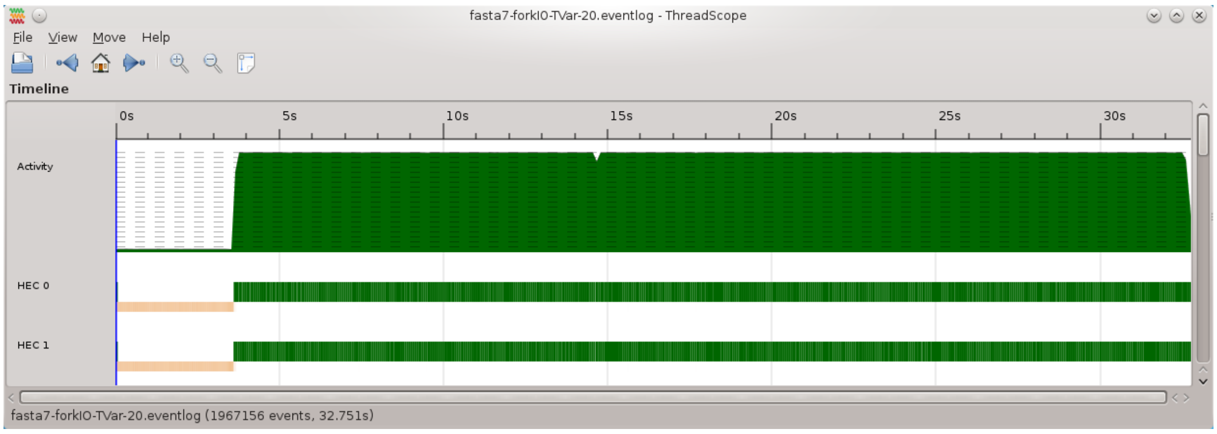
Figure 4.4: ThreadScope readings for the `forkIO-MVar` variant of `fasta`



fasta7-forkIO-MVar-20.eventlog (2029741 events, 45.737s)

Source: Made by the author

Figure 4.5: ThreadScope readings for the `forkIO-TVar` variant of `fasta`



fasta7-forkIO-TVar-20.eventlog (1967156 events, 32.751s)

Source: Made by the author

As we can see, the overall activity of the `TVar` variant is high during the whole execution while the `MVar` variant is the opposite. During profiling, the `TVar` variant is around 30% faster than the `MVar` variant, but it uses around 8x more CPU resources. These results show that,

[4]<http://ghc.haskell.org/trac/ghc/wiki/EventLog>

although the MVar variant has several worker threads, its execution is mostly sequential. This happens because, as we use an MVar to store the seed value, only one thread is generating its random numbers at a time. For the TVar variant, however, we use a TVar to store the seed value and the whole random number generation is enclosed by a transaction. In this case, the other threads are not blocked as reads to TVars are non-blocking. This leads to several threads using the same seed to generate random numbers. However, only one thread succeeds in generating these numbers because when the first one that finishes writes the new seed (seed1 from the third step of the worker's loop, Section 4.4.1) into the shared variable, the other transactions are aborted and retry.

Considering this scenario we just described, a possible explaination for the high energy consumption of the TVar variants is the transaction that encloses the random number generation being frequently aborted and re-executed. To assess this hyposesis, we have used the stm-stats[5] library. This library provides a wrapper to Haskell's atomically function that tracks the state of each transaction and counts how often the transaction was retried until it succeeded. This function is called trackNamedSTM and besides the STM action, it also receives as a parameter a String that we can use to identify each transaction. Figure 4.6 shows the output of the forkIO-TVar variant of fasta using the trackNamedSTM function instead of atomically. As we can see, the assumption is correct. Line 3 shows that, for the transaction that encloses the random number generation, there were 299 transactions that succeeded while 4138 others failed, which represents 13.84x more executions than necessary.

Figure 4.6: Output of stm-stats for forkIO-TVar variant of fasta

```
1 STM transaction statistics (2016-07-20 19:16:02.445387 UTC):
2 Transaction          Commits    Retries      Ratio
3 generate-numbers         299       4138      13.84
4 output-sync              261         33       0.13
5 wait-semaphore             2          2       1.00
```

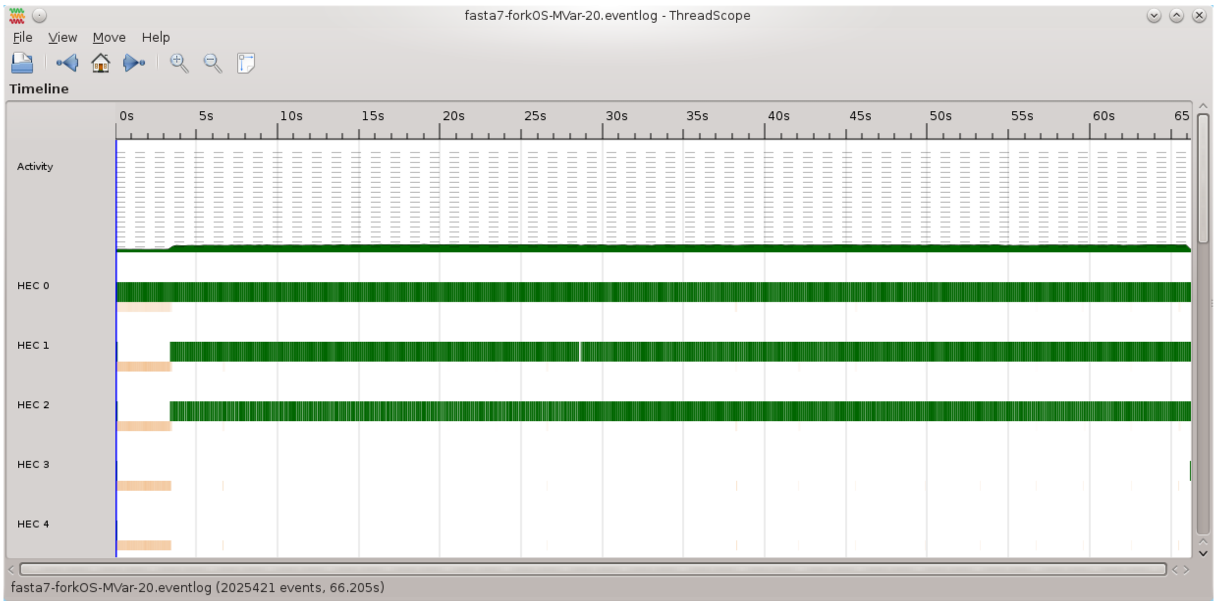
Source: Generated by stm-stats

4.4.3 The slowest consumes less energy

Another curious result is the forkOS-MVar behavior. Its execution time is considerably worse than all other variants while the energy consumption is in the middle ground. Some profiling has shown that there was no activity in some capabilities. As we can see in Figure 4.7, capabilities 0, 1, and 2 are working normally whereas capabilities 3 and 4 are idle. In this example, we executed the benchmark using 20 capabilities. The others that does not appear in this image are also iddle. This fact matches the results since having less active capabilities

[5]<http://hackage.haskell.org/package/stm-stats>

Figure 4.7: ThreadScope readings for the `forkOS-MVar` variant of `fasta`


fasta7-forkOS-MVar-20.eventlog - ThreadScope
File View Move Help
Timeline
0s
5s
10s
15s
20s
25s
30s
35s
40s
45s
50s
55s
60s
65
Activity
HEC 0
HEC 1
HEC 2
HEC 3
HEC 4
fasta7-forkOS-MVar-20.eventlog (2025421 events, 66.205s)

Source: Made by the author

implies less parallelization, which makes the program slower. However, this behavior does not seem correct. For some reason, the runtime system is scheduling the threads only for some capabilities. We also noticed that this happens only with the `forkOS-MVar` variant, changing either the thread management construct or the type of shared variable is enough for the scheduler to work as expected. We were also able to reproduce it on other machines. We observed that usually only four capabilities are used while the others stay without work.

We believe that this behaviour occurs due to a bug in Haskell's runtime system scheduler and we filed a ticket on the GHC bugtracker[6]. Simon Marlow, the mantainer and core contributor of the GHC runtime system, confirmed that he found a bug investigating this case. The bug occurs when, for instance, there are two threads on the run queue of a capability and one of them is bound to the current OS thread. In this case, the scheduler would fail to migrate any threads. He submitted a patch to fix this problem[7], which is currently under review. This patch improves parallelism in programs that have lots of bound threads. However, he also points out that after fixing this bug, the ThreadScope readings using `forkIO` and `forkOS` still do not look identical: *"even after the patch the threadscope profiles don't look identical. I don't think there is an actual problem, just that the program itself isn't very parallel - if you zoom in, there's lots of time in each thread where no work is being done. The difference in scheduling is due to the way that forkOS has to hand-shake with the new thread to get its ThreadId"*.

[6]<https://ghc.haskell.org/trac/ghc/ticket/12419>
[7]<https://phabricator.haskell.org/D2430>

4.5 Discussion

Generally, especially in the sequential benchmarks, high performance is a proxy for low energy consumption. In cooperation with our colleagues from University of Beira Interior, we conducted a complementary study (LIMA et al., 2016) that highlighted this for a number of different data structure implementations and operations. Concurrency, however, makes the relationship between performance and energy less obvious. Also, there are clear benefits in employing different thread management constructs and data-sharing primitives. This section examines this in more detail.

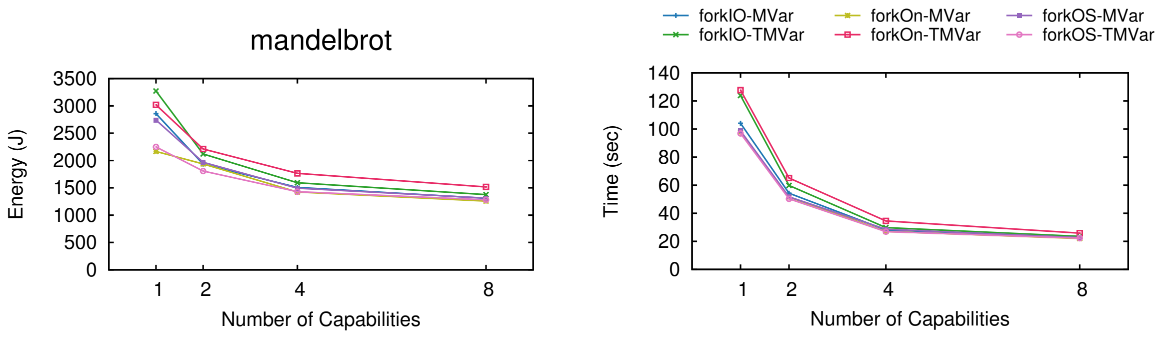
Switching between thread management construct is very simple in Haskell. Functions `forkOn`, `forkIO`, and `forkOS` take a computation of type `IO` as parameter and produce results of the same type. Thus, the only difficulty is in determining on which capability a thread created via `forkOn` will run. This is good news for developers and maintainers. Considering the seven benchmarks where we implemented variants using different data sharing primitives, in five of them the thread management construct had a stronger impact on energy usage than the data sharing primitives. Furthermore, in these five benchmarks and also in `warp` it is clearly beneficial to switch between thread management constructs.

Alternating between data sharing primitives is not as easy, but still not hard, depending on the characteristics of the program to be refactored. Going from `MVar` to `TMVar` and back is straightforward because they have very similar semantics. The only complication is that, since functions operating on `TMVar` produce results of type `STM`, calls to these functions must be enclosed in calls to `atomically` to produce a result of type `IO`. Going from `MVar` to `TVar` and back is harder, though. If a program using `MVar` does not require condition-based synchronization, it is possible to automate this transformation in a non-application-dependent manner (SOARES-NETO, 2014). If condition-based synchronization is necessary, such as is the case with the `dining-philosophers` benchmark, the semantic differences between `TVar` and `MVar` make it necessary for the maintainer to understand details of how the application was constructed.

In spite of the absence of an overall winning thread management construct or data-sharing primitive, we can identify a few cases where *a specific approach excels under specific conditions*. For instance, we can see that in both `mandelbrot` and `spectral-norm`, `forkOn` has a slightly better performance than `forkIO` and `forkOS`. In `mandelbrot`, the `forkOn` variants are around 20% more energy-efficient than the `forkIO` variants. In `spectral-norm`, `forkOn` can be up to 2x greener than `forkOS`. These two benchmarks are both CPU-intensive. They also create as many threads as the number of capabilities. In a scenario such as this, a computation-heavy algorithm with few synchronization points, keeping each thread executing in a dedicated CPU core is beneficial for the performance. This is precisely what `forkOn` does. We further explore this topic in Chapter 5.

Although there is no overall winner, for most benchmark we can point out a configuration

Figure 4.8: Energy and Time results on Alternative Platform



Source: Made by the author

that beats the others in terms of energy consumption and performance. We can observe that the ordering of the curves are more or less preserved when comparing the graph of each metric. This is amazing news for developers because: (1) small changes can make big differences; (2) it is very easy to change concurrent constructs; (3) it is cheap to experiment and perform benchmarks; and (4) it is easy to identify which configuration excels.

4.6 Threats to Validity

This work focused on the Haskell programming language. It is possible that its results do not apply to other functional programming languages, especially considering that Haskell is one of the few lazy programming languages in existence. Moreover, we analyzed only a subset of Haskell's constructs for concurrent and parallel programming. It is not possible to extrapolate the results to other constructs for concurrent and parallel execution. Nonetheless, our evaluation comprised a large number of experimental configurations that cover widely-used constructs of the Haskell language.

It is not possible to generalize the results of this study to other hardware platforms for which Haskell programs can be compiled. Factors such as operating system scheduling policies (YUAN; NAHRSTEDT, 2003) and processor and interconnect layouts (SOLERNOU et al., 2013) can clearly impact the results. We take a route common in experimental programming language research, by constructing experiments over representative system software and hardware, and the results are empirical by nature. To take a step further, we have re-executed the experiments in additional hardware configurations. The primary goal is to understand the stability and portability of our results. We ran some of the benchmarks on another machine, a 4-core Intel i7-3770 (IvyBridge) with 8 GB of DDR 1600 runing Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25) and GHC 7.10.2. Figure 4.8 shows the results of `mandelbrot` running on this i7 machine. The results show analogous trends in which the curves have similar shapes to the results of Figure 4.2. The same trend can be observed for the remaining benchmarks.

It is also not possible to generalize the results to other versions of GHC. Changes in the runtime system, for example, can lead to different results. This work also did not explore the influence of the various compiler and runtime settings of GHC. As the options range from GC algorithms to scheduling behaviour, it can have a significant impact on performance, especially for concurrency. For the benchmarks we developed, we used the default settings of GHC. For the ones from CLBG, we used the same settings used there to preserve the performance characteristics intended by the developers.

One further threat is related to our measurement approach We have employed RAPL to measure energy consumption. Thus, the results could be different for external measurement equipment. Nonetheless, previous work (HäHNEL et al., 2012) has compared the accuracy of RAPL with that of an external energy monitor and the results are consistent.

5

Guidelines for Haskell Developers

It is far, far easier to make a correct program fast than it is to make a fast program correct.

—HERB SUTTER

In this chapter, we provide guidelines for software developers to improve the energy efficiency of their concurrent Haskell programs. These guidelines are organized as suggestions where each one is composed by a brief description and a rationale. Each suggestion is presented as a section in this chapter.

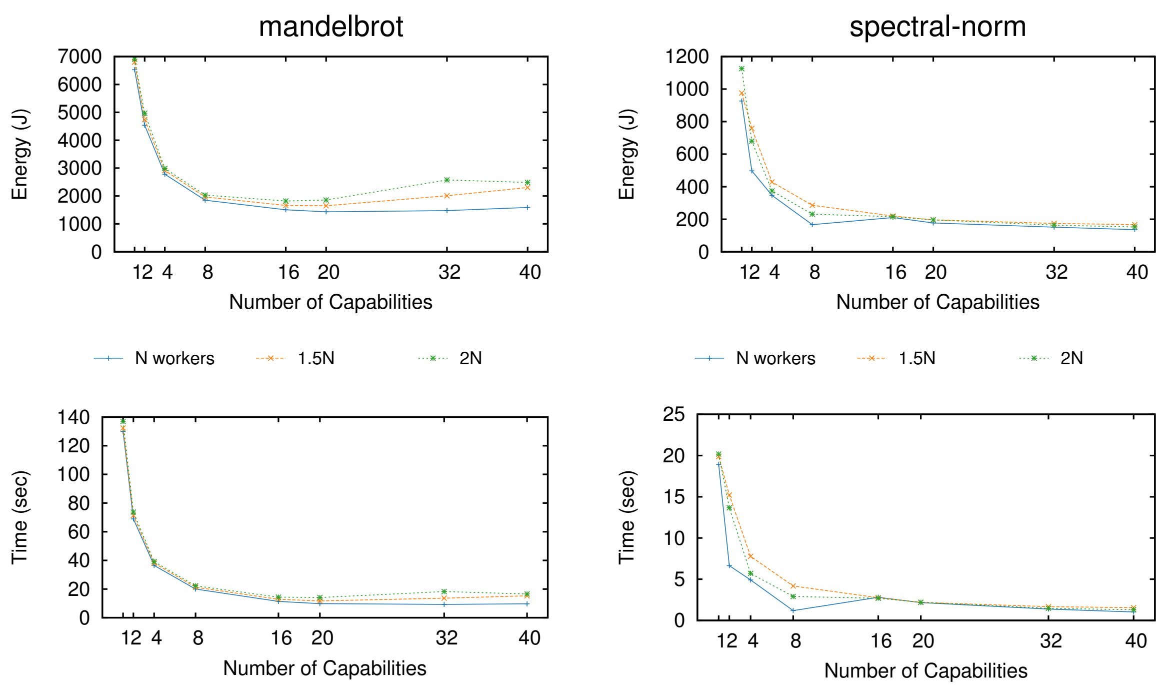
It is important to point out that these guidelines are based solely on our experimental results from Chapter 4. Both performance and energy consumption are very sensitive to the experimental conditions in which a program is executed. For this reason, developers should be aware that some contructs can behave differently on other settings. However, as we used an experimental environment with a popular and widely available architecture, we expect that our suggestions can lead to positive results in most scenarios.

5.1 Use `forkOn` for embarrassingly parallel problems

Description: Both the performance and energy consumption of a program can be improved by using `forkOn` to create new threads of execution when there is little or no dependency among these threads and they perform almost the same amount of work.

Rationale: A problem that can be decomposed into parallel tasks that do not need to communicate with each other to make progress is called *embarrassingly parallel* (HERLIHY; SHAVIT, 2012). Three of the benchmarks from Chapter 4 fit this description: `mandelbrot`, `regex-dna`, and `spectral-norm`. The results from our study has shown that, for these benchamrks, the variants using `forkOn` superseded the others in both performance and energy consumption. This result shows that manually distributing the workload in an even manner among the capabilities instead of handing this job to the runtime system scheduler improves performance. It makes sense because we know beforehand that each worker thread is doing exactly the same amount of work. In such scenarios, there is no need to migrate a thread from one capability to another since

Figure 5.1: Performence of `mandelbrot` and `spectral-norm` with different number of workers



Source: Made by the author

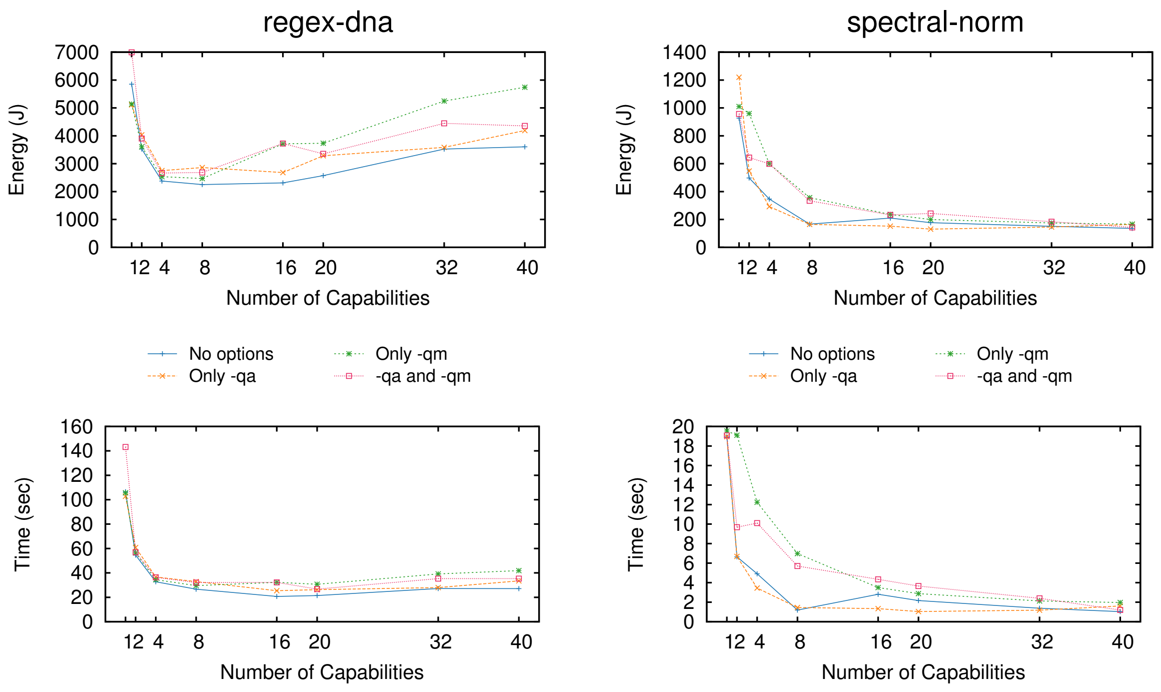
an even distribution is the one which contributes the most for the program's progress. It makes sure that each capability will have the same workload. So using `forkOn` in these cases reduces the overhead incurred by the Haskell runtime system.

However, `regex-dna` is implemented differently from `mandelbrot` and `spectral-norm`. The first one uses a fixed number of worker threads (350) while in the others the number of threads can be set by the developer. For the experiments of Chapter 4, we set these benchmarks to spawn as many threads as the number of capabilities. We decided to run another experiment with `mandelbrot` and `spectral-norm` to check how they behave if we overpopulate the capabilities' work queue. In Figure 5.1, we can see the results for the `forkOn-MVar` variant of both benchmarks with N, $1.5N$ and $2N$ worker threads, where N is the number of capabilities. As we can see, although the performance is similar, one thread per capability is the configuration with the best performance. This result makes sense because it reduces the costs of context-switching between threads of the capabilities' work queue. Thus, creating one worker thread per capability benefits both performance and energy consumption.

Additionally, there are two RTS options that, in conjunction with `forkOn`, can affect the performance of some embarrassingly parallel algorithms. The first one is the `-qa` option. It tries to pin OS threads to CPU cores using native OS facilities[1]. Using this option, the OS threads associated with a capability *i* are bound the CPU core *i*. The other one is the `-qm` option. It disables automatic migration of threads between CPUs. The former seems to fit perfectly in

[1]In Linux, GHC uses the `sched_setaffinity()` syscall

Figure 5.2: Performence of `regex-dna` and `spectral-norm` using `-qa` and `-qm`



Source: Made by the author

this context since it increases the probability of a Haskell thread being kept running on the same CPU core during the execution of the program. However, it is not clear how different the latter is from simply creating all threads with `forkOn` as we are proposing here. To get a picture of their influence, we executed our embarrassingly parallel benchmarks with these options. In Figure 5.2 we show results for the `forkOn-MVar` variant of both `regex-dna` and `spectral-norm`. Here, we executed the benchmarks without either of the options, only with `-qa`, only with `-qm`, and with both options. As we can see, the RTS options affect the performance of both benchmarks. However, the behavior is not predictable. In `spectral-norm`, using only `-qa` improves both performance and energy consumption regardless of the number of capabilities. In `regex-dna`, however, using any combination of the RTS options has a negative impact on performance. It also increases considerably the energy consumption for more than eight capabilities. We recommend developers to experiment with these options to assess how they affect the performance of a given program.

5.2 Avoid setting more capabilities than available CPUs

Description: Using more capabilities than the number of available virtual CPU cores can seriously degrade both performance and energy consumption of a concurrent Haskell program.

Rationale: A capability is thought to act as an abstraction of a CPU for the Haskell runtime system. It is the entity that can execute Haskell code. This definition implies that we can achieve maximum parallelization by creating as many capabilities as the number of CPUs. In fact, this is precisely what the official GHC documentation recommends for developers: to set `N` to be the same as the number of the processor's CPU cores. In our experiments from Chapter 4, we analyzed how each benchmark behaved for different capabilities settings. The results have shown that, for most benchmarks, the performance improved as we added more capabilities. It also confirmed the intuition that it does not make sense to outnumber the CPU cores. For eight of our benchmarks, both the performance and energy consumption were severely impaired by going from `N=40` to `N=64`.

However, modern Intel processors are equipped with a feature called *hyperthreading*. This technology increases the number of independent instructions in the processor's pipeline. For each processor core that is physically present, the operating system addresses two separate virtual cores. So from the developer's point of view, there is twice the number of CPU cores available. In this context, the GHC documentation leaves as an open question if virtual cores should be accounted: *"Whether hyperthreading cores should be counted or not is an open question; please feel free to experiment and let us know what results you find."*[2]. In our experiments, only the `spectral-norm` benchmark presented a significant improvement in both performance and energy consumption when going from `N=20` to `N=40`. All the others were negatively impacted by this change, which suggests that, in general, virtual cores should not be accounted for setting the number of capabilities.

5.3 Avoid using `forkOS` to spawn new threads

Description: Using `forkOS` undeliberately to spawn new threads of execution can degrade both performance and energy consumption of a concurrent Haskell program.

Rationale: A call to `forkOS` creates a bound thread. From a high-level perspective, it works the same way as an unbound thread created via `forkIO` or `forkOn`. They are treated as regular Haskell threads by the runtime system scheduler. However, bound threads are executed differently from the unbound ones. Each bound thread is associated with its own OS thread. So the capability has to switch OS threads when it is time to execute a bound thread. The motivation for having this kind of thread is to support interoperability with native libraries that use thread-local state. This is the scenario where the Haskell documentation recommends the use of `forkOS`. In our experiments from Chapter 4, we analyzed how each benchmark behaved if the worker threads were bound threads. The results show that none of the benchmarks benefited from using `forkOS` instead of `forkIO` or `forkOn`. Both performance and energy consumption deteriorate considerably when we use bound threads. Probably, this degradation is associated with the overhead of switching OS threads. In our benchmarks, as all threads were created using the same primitive, there is a large number of bound threads and each capability has to

[2]http://downloads.haskell.org/ ghc/7.10.2/docs/html/users_guide/using-smp.html#ftn.idp12916656

frequently switch OS threads in order to execute the scheduled action. Based on these results, we recommend developers to avoid using `forkOS` unless it is strictly required for calling foreign functions.

6

Related Work

If I have seen further it is by standing on the shoulders of giants.

—ISAAC NEWTON

In this chapter, we present a short description of the research which has been conducted in areas related to our work.

6.1 Performance Analysis in Haskell

Runciman and Wakeling (1993) introduced the first heap profiler for a lazy functional language on the Chalmers hbc/lml compiler. Later, Sansom and Peyton-Jones (1995) evolved this idea for the GHC profiler. Besides a heap profiler, they also introduced a time profiler and created the notion of cost centres. Morgan and Jarvis (1998) extended the GHC profiler with the notion of cost centre stacks for defining a hierarchy of cost centres, similar to a call graph. In this work, we take advantage of all this infrastructure provided by the GHC profiler to add energy consumption as a new profiling metric.

Jones Jr., Marlow and Singh (2009) developed a parallel profiling system for GHC. It defines a trace file format and a tool for visualizing parallel execution. This tool is called ThreadScope, and it enables developers to see a visual representation of the program execution by showing the activity of each capability and other events from the runtime system such as garbage collection. de Vries and Coutts (2014) created a tool called `ghc-events-analyze` that uses the same trace file for generating a different visualization of the program's execution. It lets developers view the CPU activity across all Haskell threads while ThreadScope shows the CPU activity across all capabilities. It also enables developers to label periods of time during program execution by instrumenting the source code with special trace calls. We did not explore these parallel profiling tools on our work. The main reason for that is that RAPL, the engine we used for collecting energy data, does not provide fine-grained energy information by cores, only the cores combined. Estimating the energy consumed by each core based on the RAPL readings is an interesting idea that we aim to tackle in the future.

Partain (1992) introduced the `nofib` benchmark suite for Haskell. This was the first attempt to build a benchmark suite for enabling a quantitative performance assessment of lazy functional programming systems. Currently, `nofib` is integrated with GHC for guaranteeing no major performance regressions in new releases. From time to time, new benchmarks are added to the suite to keep it up-to-date with the language evolution. Some of the benchmarks from The Computer Language Benchmarks Game that we employed in our study are also part of the `nofib` suite. O'Sullivan (2009b) created Criterion, a new library for measuring the performance of Haskell code. It uses a robust statistical framework for performing reliable performance analysis. We extended this tool also to work with energy consumption. We also employed it heavily in our empirical study.

6.2 Software Energy Consumption

Studying energy efficiency at the application level is an emerging direction. Traditionally, this problem has been tackled at the lower levels of the computer stack. For example, for building energy-efficient solutions for embedded software (TIWARI; MALIK; WOLFE, 1994), compilers (HSU; KREMER, 2003), operating systems (MERKEL; BELLOSA, 2006), and runtime systems (RIBIC; LIU, 2014; FARKAS et al., 2000). The programming language community has also been active researching this topic through the design of energy-aware programming languages such as Eon (SORBER et al., 2007), Green (BAEK; CHILIMBI, 2010), EnerJ (SAMPSON et al., 2011), and Energy Types (COHEN et al., 2012). In this kind of approach, the energy behavior information is encoded in the language as a first-class citizen. We take a different route in our work by trying to educate developers on writing energy-efficient software using the tools and languages that they already use.

Several related works study the impact of software changes on energy consumption. Hindle (2012) studied the effects of Mozilla Firefox's code evolution on its energy efficiency, showing a consistent reduction in energy usage correlated to performance optimizations. Pinto, Castor and Liu (2014b) studied the energy consumption of different thread management primitives in the Java programming language. We took a similar route in assessing the consumption for Haskell's thread management and data sharing constructs. Sahin, Pollock and Clause (2014) provide an analysis of the effects of code refactorings on energy consumption for nine Java applications. For six commons refactorings, such as converting local variables to fields, they showed an impact on energy consumption that was difficult to predict. Our work focuses on Haskell programs and the impact of changes regarding concurrent structures used. Those changes could be expressed as refactorings since the compared versions have the same program behavior.

Kwon and Tilevich (2013) reduced the energy consumption of mobile apps by offloading part of their computation transparently to programmers. Moura et al. (2015) studied the commit messages of 317 real-world non-trivial applications to infer the practices and needs of current application developers. A recurring theme identified in this study is the need for more tools to measure/identify/refactor energy hotspots. Manotas, Pollock and Clause (2014) described an automated support for systematically optimizing the energy usage of applications by making code-level changes. Bruce, Petke and Harman (2015) used Genetic Improvement to reduce the energy consumption of applications, reaching up to 25% reduction. Both Pinto et al. (2016) and Pereira et al. (2016) studied the energy characteristics of several collections of the Java programming language. The former analyzed it under the concurrency point-of-view, studying 16 thread-safe collections, while the latter studied other collections from the Java Collection Framework in sequential scenarios. All these approaches show the potential for program transformation, in general, and refactorings, in particular, to reduce energy consumption. We explored this potential further in this work by targeting Haskell's concurrency framework.

6.3 Refactoring

Murphy-Hill, Parnin and Black (2009) provide an analysis on the use of refactoring. Their study indicates how refactoring is common, even if only executed manually. Dig, Marrero and Ernst (2011) present some reasons why developers choose to apply program transformations to make their programs concurrent. They studied five open-source Java projects and found four categories of concurrency-related motivations for refactoring: Responsiveness, Throughput, Scalability and Correctness. Their findings show that the majority of the transformations (73.9%) consisted of modifying existing project elements, instead of creating new ones. Our work shows that modifying existing elements can also lead to energy savings, yet another motivation for refactoring.

Various papers address the problem of refactoring Haskell programs. Li, Thompson and Reinke (2005) present the Haskell Refactorer infrastructure to support the development of refactoring tools. Lee (2011) used a case study to classify 12 types of Haskell refactorings found in real projects, mostly dealing with maintainability. Brown, Loidl and Hammond (2011) specified and implemented refactorings for introducing parallelism into Haskell programs, considering mainly performance concerns. Soares-Neto (2014) described refactorings for rewriting concurrent Haskell programs to STM. Just as mentioned previously, our study may influence future Haskell program maintenance as energy efficiency becomes a mainstream concern. We are not aware of previous work analyzing the energy efficiency of Haskell programs, in particular, or purely functional programming languages, in general.

7 Conclusion

In the morning it was morning and I was still alive.
—CHARLES BUKOWSKI (Post Office)

This chapter present our concluding remarks.

7.1 Contributions

In this work, we have shedded light on the energy behavior of concurrent Haskell programs. To the best of our knowledge, this is the first attempt to analyze energy efficiency in the context of functional programming languages. Moreover, this work makes the following contributions:

- **A tool for fine-grained energy analysis.** We have extended the GHC profiler to collect and report fine-grained information about the energy consumption of a Haskell program;
- **A tool for coarse-grained energy analysis.** We have extended the Criterion microbenchmarking library to collect, perform and report statistical performance analysis of the energy consumption of Haskell code;
- **An understanding of the energy behavior of concurrent Haskell programs.** We have conducted an extensive experimental space exploration illuminating the relationship between the choices and settings of Haskell's concurrent programming constructs, and performance and energy consumption over both microbenchmarks and real-world Haskell programs;
- **A list of guidelines on how to write energy-efficient software in Haskell.** We have provided some recommendations for helping software developers to improve the energy efficiency of their concurrent Haskell programs.

An earlier version of **Chapter 4** of this dissertation has been one of the core contributions of a paper that has been published at the main research track of the *IEEE 23rd International*

Conference on Software Analysis, Evolution, and Reengineering (SANER'16) (LIMA et al., 2016). An updated version of this paper including the extensions that we made and the remainder of this dissertation is under work to be submitted to a software engineering journal.

We hope our findings will ease the development of energy-efficient Haskell programs. We also hope that this work motivates other developers and researchers from the functional programming and software engineering communities to engage in exploring the software energy consumption area.

7.2 Future Work

For improving the developers' tooling, a natural next step is enabling the GHC energy profiler to properly analyze the energy consumption of parallel programs. This involves developing a software model for estimating the energy consumed by each CPU core based on the energy consumed by all cores. This technique could also be incorporated into the parallel profiling system of GHC to add energy information to the trace file format. It would allow us to extend the ThreadScope tool to make it energy-aware. We plan to investigate this further because we believe that fine-grained energy analysis has the potential to improve significantly the developers' knowledge about software energy consumption, especially with visualization tools such as ThreadScope and `ghc-events-analyze`.

For the concurrent programming constructs, we intend to replicate our study on different hardware. Newer Intel microarchitectures such as Haswell and Broadwell have shown significant improvement over the previous generations regarding power management (HUANG et al., 2015). We want to investigate if these changes have consequences on the energy behavior of our benchmarks. We also want to explore how the various GHC options available for both the compiler and the runtime system can affect the energy consumption and performance of concurrent programs. These options enable the customization of several different aspects of the compilation and execution process, which can have a direct impact on energy consumption. Additionally, we plan to do an in-depth analysis of each benchmark of our suite to better understand its characteristics and how they differ among themselves. This can lead to the addition of new benchmarks to our suite so we can improve its diversity. Another idea under our radar is to investigate the energy behavior of other concurrent programming models that are popular in functional programming languages such as the Actor Model (AGHA, 1986).

References

AGHA, G. *Actors*: a model of concurrent computation in distributed systems. Cambridge, MA, USA: MIT Press, 1986. ISBN 0-262-01092-5.

ARMSTRONG, J. *Programming Erlang*: software for a concurrent world. [S.l.]: Pragmatic Bookshelf, 2007. ISBN 193435600X, 9781934356005.

BAEK, W.; CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI '10). *Proceedings...* New York, NY, USA: ACM, 2010. p. 198–209. ISBN 978-1-4503-0019-3.

BECKER, C. et al. Sustainability design and software: the karlskrona manifesto. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '15). *Proceedings...* Piscataway, NJ, USA: IEEE Press, 2015. p. 467–476.

BLACKBURN, S. M. et al. The dacapo benchmarks: Java benchmarking development and analysis. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA '06). *Proceedings...* New York, NY, USA: ACM, 2006. p. 169–190. ISBN 1-59593-348-4.

BROWN, C.; LOIDL, H.-W.; HAMMOND, K. Paraforming: forming parallel haskell programs using novel refactoring techniques. In: INTERNATIONAL SYMPOSIUM ON TRENDS IN FUNCTIONAL PROGRAMMING (TFP '11). *Proceedings...* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 82–97. ISBN 978-3-642-32037-8.

BRUCE, B. R.; PETKE, J.; HARMAN, M. Reducing energy consumption using genetic improvement. In: ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION (GECCO '15). *Proceedings...* New York, NY, USA: ACM, 2015. p. 1327–1334. ISBN 978-1-4503-3472-3.

BUTENHOF, D. R. *Programming with POSIX threads*. [S.l.]: Addison-Wesley Professional, 1997.

CARDELLI, L.; WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, ACM, v. 17, n. 4, p. 471–523, 1985.

CHANDRAKASAN, A. P.; SHENG, S.; BRODERSEN, R. W. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, v. 27, n. 4, p. 473–484, Apr 1992. ISSN 0018-9200.

COHEN, M. et al. Energy types. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS (OOPSLA '12). *Proceedings...* New York, NY, USA: ACM, 2012. p. 831–850. ISBN 978-1-4503-1561-6.

DAVID, H. et al. Rapl: memory power estimation and capping. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN (ISLPED '10). *Proceedings...* New York, NY, USA: ACM, 2010. p. 189–194. ISBN 978-1-4503-0146-6.

DAVISON, A. C.; HINKLEY, D. V. *Bootstrap methods and their application*. [S.l.]: Cambridge university press, 1997. v. 1.

DE VRIES, E.; COUTTS, D. *Performance profiling with ghc-events-analyze*. 2014. Disponível em: <http://www.well-typed.com/blog/86/>.

DIG, D.; MARRERO, J.; ERNST, M. D. How do programs become more concurrent: a story of program transformations. In: INTERNATIONAL WORKSHOP ON MULTICORE SOFTWARE ENGINEERING (IWMSE '11). *Proceedings...* New York, NY, USA: ACM, 2011. p. 43–50. ISBN 978-1-4503-0577-8.

DIJKSTRA, E. W. Hierarchical ordering of sequential processes. *Acta Inf.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 1, n. 2, p. 115–138, jun. 1971. ISSN 0001-5903.

DIMITROV, M. et al. *Intel® Power Governor*. 2012. Disponível em: <http://software.intel.com/en-us/articles/intel-power-governor>.

FARKAS, K. I. et al. Quantifying the energy consumption of a pocket computer and a java virtual machine. In: ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS. *Proceedings...* New York, NY, USA: ACM, 2000. p. 252–263. ISBN 1-58113-194-1.

HäHNEL, M. et al. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 40, n. 3, p. 13–17, jan. 2012. ISSN 0163-5999.

HALLER, P.; ODERSKY, M. Scala actors: unifying thread-based and event-based programming. *Theoretical Computer Science*, Elsevier, v. 410, n. 2, p. 202–220, 2009.

HARRIS, T. et al. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP '05). *Proceedings...* New York, NY, USA: ACM, 2005. p. 48–60. ISBN 1-59593-080-9.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA '93). *Proceedings...* New York, NY, USA: ACM, 1993. p. 289–300. ISBN 0-8186-3810-9.

HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123973375, 9780123977953.

HINDLE, A. Green mining: a methodology of relating software change to power consumption. In: IEEE WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR '12). *Proceedings...* Piscataway, NJ, USA: IEEE Press, 2012. p. 78–87. ISBN 978-1-4673-1761-0.

HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782.

HSU, C.-H.; KREMER, U. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In: ACM SIGPLAN 2003 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI '03). *Proceedings...* New York, NY, USA: ACM, 2003. p. 38–48. ISBN 1-58113-662-5.

HUANG, S. et al. Measurement and characterization of haswell power and energy consumption. In: INTERNATIONAL WORKSHOP ON ENERGY EFFICIENT SUPERCOMPUTING (E2SC '15). *Proceedings...* New York, NY, USA: ACM, 2015. p. 7:1–7:10. ISBN 978-1-4503-3994-0.

INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual.* [S.l.], 2016. Disponível em: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.

JONES JR., D.; MARLOW, S.; SINGH, S. Parallel performance tuning for haskell. In: ACM SIGPLAN SYMPOSIUM ON HASKELL. *Proceedings...* New York, NY, USA: ACM, 2009. p. 81–92. ISBN 978-1-60558-508-6.

KWON, Y.-W.; TILEVICH, E. Reducing the energy consumption of mobile applications behind the scenes. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM '13). *Proceedings...* [S.l.], 2013. p. 170–179. ISSN 1063-6773.

LEA, D. *Concurrent programming in Java(TM)*: design principles and patterns (3rd edition). [S.l.]: Addison-Wesley Professional, 2006. ISBN 0321256174.

LEE, D. Y. A case study on refactoring in haskell programs. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '11). *Proceedings...* New York, NY, USA: ACM, 2011. p. 1164–1166. ISBN 978-1-4503-0445-0.

LI, H.; THOMPSON, S.; REINKE, C. The haskell refactorer, hare, and its api. *Electronic Notes in Theoretical Computer Science*, v. 141, n. 4, p. 29–34, 2005. ISSN 1571-0661.

LI, P. et al. Lightweight concurrency primitives for ghc. In: ACM SIGPLAN WORKSHOP ON HASKELL WORKSHOP. *Proceedings...* New York, NY, USA: ACM, 2007. p. 107–118. ISBN 978-1-59593-674-5.

LIMA, L. G. et al. Haskell in green land: analyzing the energy behavior of a purely functional language. In: INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING (SANER '16). *Proceedings...* [S.l.], 2016. v. 1, p. 517–528.

LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 11, n. 2, p. 128–137, mar. 1977. ISSN 0163-5980.

MANOTAS, I.; POLLOCK, L.; CLAUSE, J. Seeds: a software engineer's energy-optimization decision support framework. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '14). *Proceedings...* New York, NY, USA: ACM, 2014. p. 503–514. ISBN 978-1-4503-2756-5.

MARLOW, S. Parallel and concurrent programming in haskell. In: SUMMER SCHOOL CONFERENCE ON CENTRAL EUROPEAN FUNCTIONAL PROGRAMMING SCHOOL (CEFP'11). *Proceedings...* Berlin, Heidelberg: Springer-Verlag, 2012. p. 339–401. ISBN 978-3-642-32095-8.

MERKEL, A.; BELLOSA, F. Balancing power consumption in multiprocessor systems. In: ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS (EUROSYS '06). *Proceedings...* New York, NY, USA: ACM, 2006. p. 403–414. ISBN 1-59593-322-0.

MORGAN, R. G.; JARVIS, S. A. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, Cambridge University Press, New York, NY, USA, v. 8, n. 3, p. 201–237, maio 1998. ISSN 0956-7968.

MOURA, I. et al. Mining energy-aware commits. In: IEEE WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR '15). *Proceedings...* Piscataway, NJ, USA: IEEE Press, 2015. p. 56–67.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '09). *Proceedings...* Washington, DC, USA: IEEE Computer Society, 2009. p. 287–297. ISBN 978-1-4244-3453-4.

O'SULLIVAN, B. *criterion*: robust, reliable performance measurement and analysis. 2009. Disponível em: <http://www.serpentine.com/criterion/>.

O'SULLIVAN, B. *criterion*: a new benchmarking library for haskell. 2009. Disponível em: <http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/>.

O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. *Real World Haskell*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2008. ISBN 0596514980, 9780596514983.

PANKRATIUS, V.; ADL-TABATABAI, A.-R. A study of transactional memory vs. locks in practice. In: ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES (SPAA '11). *Proceedings...* New York, NY, USA: ACM, 2011. p. 43–52. ISBN 978-1-4503-0743-7.

PARTAIN, W. The nofib benchmark suite of haskell programs. In: GLASGOW WORKSHOP ON FUNCTIONAL PROGRAMMING. *Proceedings...* [S.l.]: Springer London, 1992. p. 195–202. ISBN 978-1-4471-3215-8.

PEREIRA, R. et al. The influence of the java collection framework on overall energy consumption. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE (GREENS '16). *Proceedings...* New York, NY, USA: ACM, 2016. p. 15–21. ISBN 978-1-4503-4161-5.

PEYTON-JONES, S.; GORDON, A.; FINNE, S. Concurrent haskell. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES (POPL '96). *Proceedings...* New York, NY, USA: ACM, 1996. p. 295–308. ISBN 0-89791-769-3.

PEYTON-JONES, S. et al. The glasgow haskell compiler: a technical overview. In: JOINT FRAMEWORK FOR INFORMATION TECHNOLOGY (JFIT) TECHNICAL CONFERENCE. *Proceedings...* DTI/SERC, 1993. p. 249–257. Disponível em: <http://research.microsoft.com/apps/pubs/default.aspx?id=67076>.

PIKE, R. Go at google. In: ANNUAL CONFERENCE ON SYSTEMS, PROGRAMMING, AND APPLICATIONS: SOFTWARE FOR HUMANITY. *Proceedings...* New York, NY, USA: ACM, 2012. p. 5–6. ISBN 978-1-4503-1563-0.

PINTO, G.; CASTOR, F.; LIU, Y. D. Mining questions about software energy consumption. In: IEEE WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR '14). *Proceedings...* New York, NY, USA: ACM, 2014. p. 22–31. ISBN 978-1-4503-2863-0.

PINTO, G.; CASTOR, F.; LIU, Y. D. Understanding energy behaviors of thread management constructs. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES & APPLICATIONS (OOPSLA '14). *Proceedings...* New York, NY, USA: ACM, 2014. p. 345–360. ISBN 978-1-4503-2585-1.

PINTO, G. et al. A comprehensive study on the energy efficiency of java thread-safe collections. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME '16). *Proceedings...* [S.l.], 2016.

RIBIC, H.; LIU, Y. D. Energy-efficient work-stealing language runtimes. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS (ASPLOS '14). *Proceedings...* New York, NY, USA: ACM, 2014. p. 513–528. ISBN 978-1-4503-2305-5.

ROTEM, E. et al. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, v. 32, n. 2, p. 20–27, March 2012. ISSN 0272-1732.

RUNCIMAN, C.; WAKELING, D. Heap profiling of lazy functional programs. *Journal of Functional Programming*, v. 3, p. 217–245, 4 1993. ISSN 1469-7653.

SAHIN, C.; POLLOCK, L.; CLAUSE, J. How do code refactorings affect energy usage? In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT. *Proceedings...* New York, NY, USA: ACM, 2014. (ESEM '14), p. 36:1–36:10. ISBN 978-1-4503-2774-9.

SAMPSON, A. et al. Enerj: approximate data types for safe and general low-power computation. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI '11). *Proceedings...* New York, NY, USA: ACM, 2011. p. 164–174. ISBN 978-1-4503-0663-8.

SANSOM, P. M.; PEYTON-JONES, S. Time and space profiling for non-strict, higher-order functional languages. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES (POPL '95). *Proceedings...* New York, NY, USA: ACM, 1995. p. 355–366. ISBN 0-89791-692-1.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING (PODC '95). *Proceedings...* New York, NY, USA: ACM, 1995. p. 204–213. ISBN 0-89791-710-3.

SIVARAMAKRISHNAN, K. et al. *Composable Scheduler Activations for Haskell*. [S.l.], 2014. Disponível em: <http://research.microsoft.com/en-us/um/people/simonpj/papers/lw-conc/lwc-hs13.pdf>.

SOARES-NETO, F. *Rewriting Concurrent Haskell Programs to STM*. Dissertação (Mestrado) — Federal University of Pernambuco, February 2014.

SOLERNOU, A. et al. The effect of topology-aware process and thread placement on performance and energy. In: INTERNATIONAL SUPERCOMPUTING CONFERENCE (ISC '13). *Proceedings...* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 357–371. ISBN 978-3-642-38750-0.

SORBER, J. et al. Eon: a language and runtime system for perpetual systems. In: INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS (SENSYS '07). *Proceedings...* New York, NY, USA: ACM, 2007. p. 161–174. ISBN 978-1-59593-763-6.

SUTTER, H.; LARUS, J. Software and the concurrency revolution. *Queue*, ACM, New York, NY, USA, v. 3, n. 7, p. 54–62, set. 2005. ISSN 1542-7730.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TANI, T. *GHC(STG,Cmm,asm) illustrated for hardware persons*. 2015. Disponível em: <http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf>.

TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 2, n. 4, p. 437–445, Dec 1994. ISSN 1063-8210.

TREFETHEN, A. E.; THIYAGALINGAM, J. Energy-aware software: challenges, opportunities and strategies. *Journal of Computational Science*, v. 4, n. 6, p. 444 – 449, 2013. ISSN 1877-7503. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

VOELLMY, A. R. et al. Mio: a high-performance multicore io manager for ghc. In: ACM SIGPLAN SYMPOSIUM ON HASKELL. *Proceedings...* New York, NY, USA: ACM, 2013. p. 129–140. ISBN 978-1-4503-2383-3.

WEAVER, V. M. et al. Measuring energy and power with papi. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS (ICPPW '12). *Proceedings...* [S.l.], 2012. p. 262–268. ISSN 0190-3918.

YUAN, W.; NAHRSTEDT, K. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP '03). *Proceedings...* New York, NY, USA: ACM, 2003. p. 149–163. ISBN 1-58113-757-5.

Appendix

A

The `spectral-norm` Benchmark

This benchmark is part of The Computer Language Benchmarks Game suite. It is based on a MathWorld challenge called *"Hundred-Dollar, Hundred-Digit Challenge Problems"* by Eric W. Weisstein[1]. To submit a solution for this problem on The Computer Language Benchmarks Game, the program should not only give the correct result, but also use the same algorithm to calculate that result. Each program should:

1. Calculate the spectral norm of an infinite matrix A, with entries $a_{11} = 1, a_{12} = \frac{1}{2}, a_{21} = \frac{1}{3}, a_{13} = \frac{1}{4}, a_{22} = \frac{1}{5}, a_{31} = \frac{1}{6} \ldots$

2. Implement 4 separate functions to:

   - Return element (i, j) of infinite matrix A
   - Multiply vector V by matrix A
   - Multiply vector V by matrix A transposed
   - Multiply vector V by matrix A and then by matrix A transposed

Following, we show the implementation we used for this benchmark. In this case, we are showing only the source code for the `forkIO` variant as the refactoring to change the thread management construct is straightforward. We split it in multiple sections to better present it here, but it can be seen as a whole on GitHub[2]. Code A.1 is the entry point of the program where we define the Criterion benchmark that will call `spectral-norm`. Code A.2 and Code A.3 are common parts that were not refactored. Code A.4, Code A.5 and Code A.6 show the implementation of the `CyclicBarrier` type for the `MVar`, `TMVar` and `TVar` variants, respectively. Finally, Code A.7 shows the fours functions required by the problem statement.

[1]http://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html
[2]https://github.com/green-haskell/concurrency-benchmark

Code A.1: The Criterion benchmark

```haskell
import Criterion.Main
import Program as S (exec)

main :: IO ()
main = defaultMain [
    bench "spectral-norm #4" $ nfIO (S.exec 10000)
  ]
```

Code A.2: Entry point of `spectral-norm`

```haskell
module Program (exec,main) where

import Utils(redirectStdOutAndRun)

import System.Environment
import Foreign.Marshal.Array
import Foreign
import Text.Printf
import Control.Concurrent
import Control.Monad
import GHC.Base hiding (foldr)
import GHC.Conc
import System.IO
import System.Directory (removeFile,getTemporaryDirectory)
import GHC.IO.Handle

type Reals = Ptr Double

main = do
    n <- getArgs >>= readIO . head
    exec' n

exec n = redirectStdOutAndRun exec' n

exec' n = do
    allocaArray n $ \ u -> allocaArray n $ \ v -> do
      forM_ [0..n-1] $ \i -> pokeElemOff u i 1 >> pokeElemOff v i 0

      powerMethod 10 n u v
      printf "%.9f\n" =<< eigenvalue n u v 0 0 0
```

Code A.3: A function to calculate the eigenvalue of a matrix

```haskell
eigenvalue :: Int -> Reals -> Reals -> Int -> Double -> Double -> IO Double
eigenvalue !n !u !v !i !vBv !vv
    | i < n     = do        ui <- peekElemOff u i
                        vi <- peekElemOff v i
                        eigenvalue n u v (i+1) (vBv + ui * vi) (vv + vi * vi)
    | otherwise = return $! sqrt $! vBv / vv
```

Code A.4: `CyclicBarrier` defined for the `MVar` variant

```haskell
data CyclicBarrier = Cyclic !Int !(MVar (Int, [MVar ()]))

await :: CyclicBarrier -> IO ()
await (Cyclic k waitsVar) = do
        (x, waits) <- takeMVar waitsVar
        if x <= 1 then do
                mapM_ (`putMVar` ()) waits
                putMVar waitsVar (k, [])
          else do
                  var <- newEmptyMVar
                  putMVar waitsVar (x-1,var:waits)
                  takeMVar var

newCyclicBarrier :: Int -> IO CyclicBarrier
newCyclicBarrier k = liftM (Cyclic k) (newMVar (k, []))
```

Code A.5: `CyclicBarrier` defined for the `TMVar` variant

```haskell
data CyclicBarrier = Cyclic !Int !(TMVar (Int, [TMVar ()]))

await :: CyclicBarrier -> IO ()
await (Cyclic k waitsVar) = do
        (x, waits) <- atomically $ takeTMVar waitsVar
        if x <= 1 then do
                mapM_ (\x -> atomically $ putTMVar x ()) waits
                atomically $ putTMVar waitsVar (k, [])
          else do
                  var <- newEmptyTMVarIO
                  atomically $ putTMVar waitsVar (x-1,var:waits)
                  atomically $ takeTMVar var

newCyclicBarrier :: Int -> IO CyclicBarrier
newCyclicBarrier k = liftM (Cyclic k) (newTMVarIO (k, []))
```

Code A.6: `CyclicBarrier` defined for the `TVar` variant

```haskell
data CyclicBarrier = Cyclic !Int !(TVar (Int, [TMVar ()]))

await :: CyclicBarrier -> IO ()
await (Cyclic k waitsVar) = join $ atomically $ do
        (x, waits) <- readTVar waitsVar
        if x <= 1 then do
                mapM_ (\x -> putTMVar x ()) waits
                writeTVar waitsVar (k, [])
                return $ return ()
          else do
                  var <- newEmptyTMVar
                  writeTVar waitsVar (x-1,var:waits)
                  return $ atomically $ takeTMVar var

newCyclicBarrier :: Int -> IO CyclicBarrier
newCyclicBarrier k = liftM (Cyclic k) (atomically $ newTVar (k, []))
```

Code A.7: The four functions to manipulate the matrix

```haskell
powerMethod :: Int -> Int -> Reals -> Reals -> IO ()
powerMethod z n u v = allocaArray n $ \ !t -> do
        let chunk = (n + numCapabilities - 1) `quotInt` numCapabilities
        !barrier <- newCyclicBarrier $! (n + chunk - 1) `quotInt` chunk
        let timesAtAv !s !d l r = do
                timesAv n s t l r
                await barrier
                timesAtv n t d l r
                await barrier
        let thread !l !r = foldr (>>) (return ()) $ replicate z $ do
                timesAtAv u v l r
                timesAtAv v u l r
        let go l = case l + chunk of
                r | r < n        -> forkIO (thread l r) >> go r
                  | otherwise        -> thread l n
        go 0

timesAv :: Int -> Reals -> Reals -> Int -> Int -> IO ()
timesAv !n !u !au !l !r = go l where
    go :: Int -> IO ()
    go !i = when (i < r) $ do
        let avsum !j !acc
                | j < n = do
                        !uj <- peekElemOff u j
                        avsum (j+1) (acc + ((aij i j) * uj))
                | otherwise = pokeElemOff au i acc >> go (i+1)
        avsum 0 0

timesAtv :: Int -> Reals -> Reals -> Int -> Int -> IO ()
timesAtv !n !u !a !l !r = go l
  where
    go :: Int -> IO ()
    go !i = when (i < r) $ do
        let atvsum !j !acc
                | j < n        = do        !uj <- peekElemOff u j
                                atvsum (j+1) (acc + ((aij j i) * uj))
                | otherwise = pokeElemOff a i acc >> go (i+1)
        atvsum 0 0

--
-- manually unbox the inner loop:
-- aij i j = 1 / fromIntegral ((i+j) * (i+j+1) `div` 2 + i + 1)
--
aij (I# i) (I# j) = D# (
    case i +# j of
        n -> 1.0## /## int2Double#
                (((n *# (n+#1#)) `uncheckedIShiftRA#` 1#) +# (i +# 1#)))
```