The Journal of Systems and Software 149 (2019) 554–580

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On Haskell and energy efficiency

Luís Gabriel Lima[a], Francisco Soares-Neto[b], Paulo Lieuthier[c], Fernando Castor[d], Gilberto Melfe[e,*], João Paulo Fernandes[e]

[a] Amazon Web Services
[b] Apple Developer Academy, UFPE
[c] In Loco
[d] Informatics Center Federal University of Pernambuco (UFPE) Recife, Brazil
[e] CISUC Department of Informatics Engineering University of Coimbra Coimbra, Portugal

A R T I C L E I N F O

Article history:
Received 3 April 2018
Revised 16 November 2018
Accepted 16 December 2018
Available online 18 December 2018

Keywords:
Energy efficiency
Functional programming
Haskell

A B S T R A C T

Background: Recent work has studied diverse affecting factors on software energy efficiency.

Objective: This paper attempts to shed light on the energy behavior of programs written in a lazy, purely functional programming language, Haskell.

Methodology: We conducted two in-depth and complementary studies to analyze the energy efficiency of programs from two different perspectives: strictness and concurrency.

Results: We found that small changes can make a big difference. In one benchmark, under a specific configuration, choosing the MVar data sharing primitive over TMVar can yield 60% energy savings. In another benchmark, TMVar can yield up to 30% savings over MVar. Thus, tools that support developers in refactoring a program to switch between primitives can be very useful. In addition, the relationship between energy consumption and performance is not always clear. In sequential benchmarks, high performance is an accurate proxy for low energy consumption. However, for one of our concurrent benchmarks, the variants with the best performance also exhibited the worst energy consumption. We report on deviating cases.

Conclusions: To support developers, we have extended existing performance analysis tools to also gather and present data about energy consumption. Furthermore, we provide a set of guidelines to help Haskell developers save energy.

1. Introduction

Energy-efficiency has concerned hardware and low-level software engineers for years (Chandrakasan et al., 1992; Tiwari et al., 1994; Yuan and Nahrstedt, 2003). However, the growing worldwide movement towards sustainability, including sustainability in software (Becker et al., 2015), combined with the systemic nature of energy efficiency as a quality attribute have motivated the study of the energy impact of application software in execution. This tendency has led researchers to evaluate existing techniques, tools, and languages for application development from an energy-centric perspective. Recent work has studied the effect that factors such as code obfuscation (Sahin et al., 2014b), Android API calls (Vásquez et al., 2014), programming languages (Oliveira et al., 2017; Pereira et al., 2017), object-oriented code refactorings (Sahin et al., 2014a), constructs for concurrent execution (Pinto et al., 2014), and data types (Liu et al., 2015) have on energy efficiency. Analyzing the impact of different factors on energy is important for software developers and maintainers. It can inform their decisions about the best and worst solution for a particular context. Moreover, it is important to make developers aware that seemingly small modifications can yield considerable gains in terms of energy. For example, a study by Vásquez et al. (2014) has discovered that some Android API calls consume 3000 times more energy than the average Android API call. These API calls should clearly be avoided if energy is an important requirement.

In this paper, we explore an additional dimension. We attempt to shed light on the energy behavior of programs written in a lazy, purely functional language. More specifically, we target pro-

* Corresponding author.
E-mail addresses: luislim@amazon.com (L.G. Lima), francisco@academy.cin.ufpe.br (F. Soares-Neto), paulo.vitor@inloco.com.br (P. Lieuthier), castor@cin.ufpe.br (F. Castor), gilbertomelfe@gmail.com (G. Melfe), jpf@dei.uc.pt (J.P. Fernandes).

https://doi.org/10.1016/j.jss.2018.12.014
0164-1212/© 2018 Elsevier Inc. All rights reserved.

grams written in Haskell.[1] Functional languages, in general, include a number of features that are not generally available in imperative programming languages. In particular, Haskell has mature implementations of sophisticated features such as laziness, partial function application, software transactional memory, tail recursion, and a kind system (Pierce, 2002). Furthermore, recursion is the norm in Haskell programs and side effects are restricted by the type system of the language. Due to all these differences, it is possible that programs written in such a language behave differently from those written in imperative languages, from an energy perspective. Additionally, functional languages and features are increasing in popularity. Huge corporations with concerns for energy consumption, such as Facebook, use Haskell for efficient parallel data access on their servers (Marlow et al., 2014). Meanwhile, mainstream programming languages like Java and C# have adopted functional programming features such as lambdas (Corporation, 2018; Hejlsberg and Torgersen, 2018).

We analyze the energy efficiency of Haskell programs from two different perspectives: strictness and concurrency.

By default, expressions in Haskell are lazily evaluated, meaning that any given expression will only be evaluated when it is first necessary. This is different from most programming languages, where expressions are evaluated strictly and possibly multiple times. In Haskell, it is possible to force strict evaluation in contexts where this is useful. This is very important to analyze the performance and energy efficiency of Haskell programs. Indeed, to analize a program, one needs to ensure that all the computational work whose efficiency we want to assess is actually performed, i.e., that such work is not avoided by lazy evaluation (Apfelmus, 2014; 2015).

As for concurrency, previous work (Pinto et al., 2014; Trefethen and Thiyagalingam, 2013) has demonstrated that concurrent programming constructs can influence energy consumption in unforeseen ways.

In this paper, we attempt to shed more light on these complex subjects. More specifically, we address the following high-level research question:

RQ. To what extent is it possible to improve the performance and energy efficiency of Haskell programs by using different data structure implementations or concurrent programming constructs?

To gain insight into the answer to this question, we conducted two in-depth, complementary empirical studies. In the first study, we analyzed the performance and energy behavior of several benchmark operations over 16 different implementations of four different types of data structures. Even though Haskell has several implementations of well-known data structures (Okasaki, 2001), we are not aware of any experimental evaluation of these implementations. In the second one, we assessed three different thread management constructs and three primitives for data sharing using nine benchmarks and multiple experimental configurations. The work reported in this paper extends work we have conducted and reported previously (Lima et al., 2016). Later in this section we discuss how it expands on our previous work.

We found that small changes can make a big difference in terms of energy consumption. For example, in one of our benchmarks, under a specific configuration, choosing one data sharing primitive (MVar) over another (TMVar) can yield 60% energy savings. Nonetheless, there is no universal winner. The results vary depending on the characteristics of each program. In another benchmark, TMVars can yield up to 30% energy savings over MVars. Thus, tools that support developers in quickly refactoring a program to switch between different primitives can be of great help if energy is a concern. In addition, the relationship between energy consumption and performance is not always clear. Generally, especially in the sequential benchmarks, high performance is a proxy for low energy consumption, though some exceptions do exist. Nonetheless, when concurrency comes into play, we found scenarios where the configuration with the best performance (30% faster than the one with the worst performance) also exhibited the second worst energy consumption (used 133% more energy than the one with the lowest usage).

To support developers in better understanding this complex relationship, we have extended two existing tools for performance analysis to make them *energy-aware*. The first one is the Criterion benchmarking library, which we have employed extensively in the two studies. The second one is the profiler that comes with the Glasgow Haskell Compiler. The data for this study, as well as the source code for the implemented tools and benchmarks can be found at green-haskell.github.io.

This paper expands on our previous work (Lima et al., 2016) in a number of ways. First, it provides an in-depth analysis of the behavior of the purely functional data structures. In particular, we conducted experiments that varied the base size of the studied data structures, profiled their execution, and analyzed the underlying causes for their behavior. Furthermore, we conducted an in-depth analysis of with one of the concurrent benchmarks, fasta. Finally, based on our experimental results, we provide practical guidelines for Haskell developers to save energy. To the best of our knowledge, no such guidelines currently exist in the Haskell literature. Both the fasta analysis and the practical guidelines first appeared in a master's thesis (Lima, 2016).

2. A Haskell Primer

Haskell[2] 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 Listing 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.

[1] To be rigorous, the specification of the Haskell language only specifies that a compiler needs to be non-strict, but since in our work we use the Glasgow Haskell Compiler, which makes essential use of lazyness, in this paper we will refer to Haskell as a lazy language.

[2] This section uses text that appeared first in a master's dissertation (Lima, 2016).

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

Listing. 1. A recursive factorial function.

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

Listing. 2. A polymorphic function to reverse a list.

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

Listing. 3. The `map` function.

```haskell
data Tree t = Node t (Tree t) (Tree t)
            | Empty

depth :: Tree t -> Int
depth Empty        = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

Listing. 4. A data type for binary tree and a function to calculate its depth.

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++. Listing 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 and Wegner, 1985).

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 Listing 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.

In Haskell, a developer can also extend the built-in primitive types by defining new *abstract data types*. Listing 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.

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 Listing 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.

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 Listing 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 5.1. In Listing 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, 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.

Finally, as can be seen in Listing 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.

3. Measuring energy consumption

This section presents the technology we used to measure the energy consumption of `Haskell` programs. In Section 3.1, we give a brief overview of our interface to gather energy information from Intel processors. In Sections 3.2 and 3.3 we explain how we extended existing `Haskell` performance analysis tools to also work with energy consumption.

All experiments presented in this paper were conducted on a machine with 2x10-core Intel Xeon E5-2660 v2 processors (Ivy Bridge microarchitecture, 2-node NUMA) and 256GB of DDR3 1600MHz memory. This machine runs the Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25) OS. The compiler was GHC 7.10.2, using Edison 1.3 (Section 4.1), and a modified Criterion (Section 3.2) library. Also, all experiments were performed with no other load on the OS.

3.1. RAPL

Running Average Power Limit (RAPL) (David et al., 2010) is an interface provided by modern Intel processors to allow setting custom power limits to the processor packages. Using this interface one can access energy and power readings via a model-specific register (MSR). RAPL uses a software power model to estimate the energy consumption based on various hardware performance counters, temperature, leakage models and I/O models (Weaver et al., 2012).

The precision and reliability of RAPL have been extensively studied (Rotem et al., 2012; Hähnel et al., 2012; Desrochers et al., 2016), showing that, although there is in general an offset between RAPL estimations and the corresponding physically measured values, the general behavior over time is consistent between the two observations. And, for server machines, which are the ones we target, this offset is actually insignificant.

```
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)
```

Listing. 5. Definition of an `Eq` typeclass instance for the `Tree` data type.

```
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
```

Listing. 6. Definition of type class `Monad` .

```
-- 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!")
```

Listing. 7. Basic `IO` example.

RAPL interfaces operate at the granularity of a processor socket (package). There are MSRs to access 4 domains:

- PKG: total energy consumed by an entire socket
- PP0: energy consumed by all cores and caches
- PP1: energy consumed by the on-chip GPU
- DRAM: energy consumed by all DIMMs

The client platforms have access to {PKG, PP0, PP1} while the server platforms have access to {PKG, PP0, DRAM}. For this work, we collected the energy consumption data from the PKG domain using the `msr` module of the Linux kernel to access the MSR readings.

3.2. *Criterion*

`Criterion` (O'Sullivan, 2009) is a microbenchmarking library that is used to measure the performance of `Haskell` code. It provides a framework for both the execution of the benchmarks and the analysis of their results, being able to measure events with duration in the order of picoseconds.

`Criterion` is robust enough to filter out noise coming, e.g., from the clock resolution, the operating system's scheduling or garbage collection. `Criterion`'s strategy to mitigate noise is to measure many runs of a benchmark in sequence and then use a linear regression model to estimate the time needed for a single run. In this manner, the outliers become visible. Having been proposed in the context of a functional language with lazy evaluation, `Criterion` natively offers mechanisms to evaluate the results of a benchmark in different *depths*, such as weak head normal form and normal form. `Criterion` is able to measure CPU time, CPU cycles, memory allocation and garbage collection. In our work, we have extended its domain so that it is also able to measure the amount of energy consumed during the execution of a benchmark.

The adaptation of `Criterion` has been conducted based on two essential considerations. First, the energy consumed in the sampling time intervals used by `Criterion` is obtained via external C function invocations to RAPL. This is similar to the time measurements natively provided by `Criterion`, which are also realized via foreign function interface (FFI) calls. Second, we need to handle possible overflows occurring on RAPL registers (David et al., 2010). For two consecutive reads `x` and `y` of values in such registers, this was achieved by discarding the energy consumed in the corresponding (extremely small) time interval if `y`, which is read later, is smaller than `x`.

In the extended version of `Criterion`, energy consumption is measured in the same execution of the benchmarks which is used to measure runtime performance. In this version, all the aforementioned aspects of `Criterion`'s original methodology have straightforwardly been adapted for energy consumption analysis.

3.3. *GHC profiler*

Currently, the GHC profiler is capable of measuring time and space usage. Developers can enable profiling by compiling a program with the `-prof` flag. This makes the final executable hold the profiling routines as part of the runtime system. Then, when running this executable passing the `+RTS -p` argument, the runtime system will collect data from the execution to produce a report at the end. This report contains fine-grained information of both time and space usage for each cost center. The cost centers can be manually added to the source code by the developer or automatically generated by the compiler.

To extend the profiler to also collect energy consumption data, we based our solution on the approach used by the time profiler (Sansom and Peyton Jones, 1995). To measure the execution time, the profiler keeps in each cost center a tick counter. At any moment, the cost center that is currently executing is held in a special register by the runtime system. Then, a regular clock interrupt runs a routine to increment this tick counter of the current cost center. This makes it possible to determine and report the relative cost of the different parts of the program.

Similarly, the energy profiler keeps in each cost center an accumulator. At each clock interrupt, the profiler adds to the accumulator of the current cost center the energy consumed between the previous and current interrupt. At the end of the execution, it can report the fine-grained information of energy consumption for each cost center.

4. Comparing purely functional data structures

In this section, we present our scenario to analyze and compare different implementations for concrete data abstractions. Our study is motivated by the following research questions:

RQ1. How do different *implementations* of the same abstractions compare in terms of runtime and energy efficiency?
RQ2. For concrete operations, what is the relationship between their *performance* and their energy consumption?

In Section 4.1, we describe a general purpose library providing different implementations for abstractions such as `Sequences` or `Collections`. In order to establish a comparison baseline, these implementations were exercised by the series of operations defined in the benchmark described in Section 4.2.

Table 1
Abstractions and implementations available in Edison.

Collections	Associative collections	Sequences
		BankersQueue
EnumSet		SimpleQueue
StandardSet		BinaryRandList
UnbalancedSet	AssocList	JoinList
LazyPairingHeap	PatriciaLoMap	RandList
LeftistHeap	StandardMap	BraunSeq
MinHeap	TernaryTrie	FingerSeq
SkewHeap		ListSeq
SplayHeap		RevSeq
		SizedSeq
		MyersStack

Table 2
Benchmark operations.

Iters	Operation	Base	Elems
1	add	100,000	100,000
1000	addAll	100,000	1000
1	clear	100,000	n.a.
1000	contains	100,000	1
5000	containsAll	100,000	1000
1	iterator	100,000	n.a.
10000	remove	100,000	1
10	removeAll	100,000	1000
10	retainAll	100,000	1000
5000	toArray	100,000	n.a.

4.1. A library of purely functional data structures

Our analysis relies on Edison, a mature and well documented library of purely functional data structures (Okasaki, 2001; 1999). Edison provides different functional data structures for implementing three types of abstractions: Sequences, Collections, and Associative Collections. While these implementations are available in other programming languages, e.g., in ML (Okasaki, 1999), here we focus on their Haskell version. While this version already incorporates an extensive unit test suite to guarantee functional correctness, it can admittedly benefit from the type of performance analysis we consider here (Dockins, 2018a).

In Table 1, we list all the implementations that are available for each of the abstractions considered by Edison. These implementations can also be consulted in the EdisonCore (Dockins, 2018c) and EdisonAPI (Dockins, 2018b) packages. Some of the listed implementations are actually *adaptors*. This is the case, e.g., of SizedSeq that adds a size parameter to any implementation of Sequences. Besides SizedSeq, also RevSeq, for Sequences, and MinHeap for Collections are adaptors for other implementations.

We do not present here the complete lists of functions in the respective APIs, but this information is available at green-haskell.github.io.

4.1.1. Collections

The Collections abstraction includes sets and heaps. While all implementations of these data structures share a significant amount of (reusable and uniform) functions, there are also functions that are specific of sets. The intersection method, for example, operates over two sets and conceptually only makes sense considering the uniqueness of elements in each set. A restriction such as this does not make sense to assume for heaps.

4.1.2. Associative Collections

The associative collections abstraction includes finite maps and finite relations. They generically map keys of type k to values of type a. Exceptions are the PatriciaLoMap and TernaryTrie implementations which use more restricted types of keys (Int and [k] respectively). All other implementations respect the same API.

4.1.3. Sequences

In Edison, the Sequences abstraction includes, e.g., lists, queues and stacks. Furthermore, all implementations of the Sequence abstraction define a reusable, coherent and uniform set of functions.

4.2. Benchmark

Following the approach considered in different studies (Manotas et al., 2014; Carção, 2014; Pinto et al., 2016), our benchmark is inspired by the microbenchmark to evaluate the run time performance of Java's JDK Collection API implementations (Lewis, 2011). The operations we consider are listed in Table 2, and they all can be abstracted by the format:

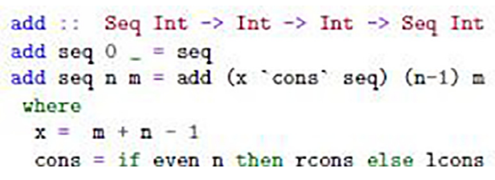
$$iters * operation(base, elems)$$

This format reads as: iterate *operation* a given number of times (*iters*) over a data structure with a *base* number of elements. If *operation* requires an additional data structure, the number of elements in it is given by *elems*. All the operations are suggested to be executed over a base structure with 100000 elements. So, e.g., for the addAll operation, the second entry in the table suggests adding 1000 times all the elements of a structure with 1000 elements to the base structure (of size 100000).

4.3. Methodology

Our analysis proceeded by applying the benchmark defined in the previous section to the different implementations provided by Edison. For different reasons, we ended up excluding some implementations from our experimental setting. This was the case of RevSeq and SizedSeq, for Sequences, and MinHeap for Heaps, since they are adaptors of other implementations for the corresponding abstractions. EnumSet, for Sets, was not considered because it can only hold a limited number of elements, which makes it not compatible with the considered benchmark. As said before, PatriciaLoMap and TernaryTrie are not totally compatible with the Associative Collections API, so they could not be used in our uniform benchmark. Finally, MyersStack, for Sequences was discarded since its underlying data structure has redundant information in such a way that fully evaluating its instances has exponential behavior. We have also split the comparison of Collections in independent comparisons of Heaps and Sets. This is due to the fact that these abstractions do not strictly adhere to the same API.

Table 3 presents the complete list of Edison functions that were used in the implementation of the benchmark operations. Most operations in the underlying benchmark have straightforward correspondences in the implementation functions provided by Edison. This is the case, for example, of the operation add, which can naturally be interpreted by functions insert, for Heaps, Sets and Associative Collections. For Sequences, the underlying ordering notion allows two possible interpretations for adding an element to a sequence: in its beginning or in its end. In this case, we defined add as follows, to alternatively use both interpretations:



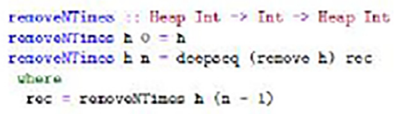
```
add :: Seq Int -> Int -> Int -> Seq Int
add seq 0 _ = seq
add seq n m = add (x `cons` seq) (n-1) m
 where
  x = m + n - 1
  cons = if even n then rcons else lcons
```

Table 3
Edison functions used to implement the benchmark operations.

	Sequences	Sets	Heaps	Associative Collections
add	lcons, rcons	insert	insert	insert
addAll	append	union	union	union
clear	null, ltail	difference	minView, delete	difference
contains	null, filter	member	member	member
containsAll	foldr, map	subset	null, member, minView	submap
iterator	map	foldr	fold	map
remove	null, ltail	deleteMin	deleteMin	null, deleteMin
removeAll	filter	difference	minView, delete	difference
retainAll	filter	intersection	filter, member	intersection- With
toArray	toList	foldr	fold	foldrWithKey

With the previous definition, `add s n m` inserts the `n` elements `{n+m-1,n+m-2, ..., m}` to `s`.

In the context of a language with lazy evaluation such as `Haskell`, the operations that the benchmark suggests to iterate a given number of times need to be implemented carefully, in a way that ensures that the result of each iteration is fully evaluated. Indeed, while the full evaluation of the final result can be ensured by the use of `Criterion`, if the intermediate ones are not demanded, the lazy evaluation does not build them. This led us to use primitives such as `deepseq` (deepseq package, 2018) in many definitions. This primitive forces the strict evaluation of an expression, ignoring Haskell's laziness. We present an example below, where we employed `deepseq` to iterate a number of times the `remove` operation for `Heaps`.



```
removeNTimes :: Heap Int -> Int -> Heap Int
removeNTimes h 0 = h
removeNTimes h n = deepseq (remove h) rec
 where
  rec = removeNTimes h (n - 1)
```

We have tried to follow as much as possible the data structure sizes suggested by the benchmark described in the previous section. The data structure creation relies on the insertion function(s) that is(are) available in the APIs of the respective abstractions. Our (admittedly simple) strategy to create a data structure of size n was to insert, in this order, the values {n-1, n-2, ., 0} into an empty structure. Although simple, this strategy allowed us to control the elements present in a data structure: this was used, e.g., in search operations, to ensure that structures were fully traversed (by searching for an element that is guaranteed not to be present).

We have also tried to follow the numbers of iterations suggested by the benchmark.

In a few cases, however, we needed to simplify concrete operations for specific abstractions. This simplification was performed whenever a concrete operation failed to terminate within a 3 hours bound for a given implementation. In such cases, we repeatedly halved the size of the base data structure, starting at 100000, 50000 and so on. When the data structure size of 3125 was reached without the bound being met, we started halving the number of iterations. With this principle in mind, no change was necessary for `Heaps` and `Sets`. For `Associative Collections` and `Sequences`, however, this was not the case. Table 4 lists the operations whose inputs or numbers of iterations were simplified. The underlined elements of this table are the ones that differ from the original benchmark.

The **independent variables** of this study are data structure implementations (16), operations (10), and base data structure sizes (5). Combinations of these variables amount to 800 experimental configurations. The **dependent variables** of the study are execution time and energy consumption.

Table 4
Modified benchmark operations.

Abstraction	Iters	Operation	Base	Elems
Associative Collections	1	clear	50000	n.a.
	2500	remove	3125	1
	10	retainAll	25000	1000
	2500	toArray	3125	n.a.
Sequences	1	add	3125	50000
	625	containsAll	3125	1000

Table 5
Correlation between time and energy consumption for the analyzed abstractions.

Abstraction	Spearman correlation	p-value
Sets	1	2.2e−16
Heaps	0.9993902	$<$ 2.2e−16
Associative Collections	1	$<$ 5.976e−06
Sequences	0.9999531	$<$ 2.2e−16

4.4. Results

We analyze the results we obtained following the methodology described in the previous section. Here, we present charts that are representative of the general observations, but all charts for all operations on all abstractions are available at the companion website, at green-haskell.github.io.

We have confirmed that our analyses in the remainder of this section are statistically valid, by calculating correlation coefficients given by Spearman's non parametric measure. Indeed, we studied the correlation between execution time and energy consumption within each of the 4 abstractions that we considered. For this, we calculated 4 correlation coefficients, considering in each two data series: (i) the execution time and (ii) the energy consumption, for all the operations within the respective abstraction. We found that these variables are strongly correlated, which is indicated by the correlation coefficients and respective p-values given in Table 5.

Fig. 1 uses scatterplots to give a graphical perspective on the relationship between energy and time for each of the 4 abstractions that we considered.

Sets. We have observed that for each combination of implementation and benchmark operation, taking longer to execute also implies more energy consumption. The `UnbalancedSet` implementation is less efficient (both in terms of runtime and energy footprint) than `StandardSet` for all benchmark operations except `contains`.

The results on the comparison between both implementations for the `clear` operation of the benchmark are presented in Fig. 2. In Figs. 2(a) and (b) we compare the absolute values obtained for the runtime execution and energy consumption, respectively. In Fig. 2(c) we compare the proportions of time and energy consump-

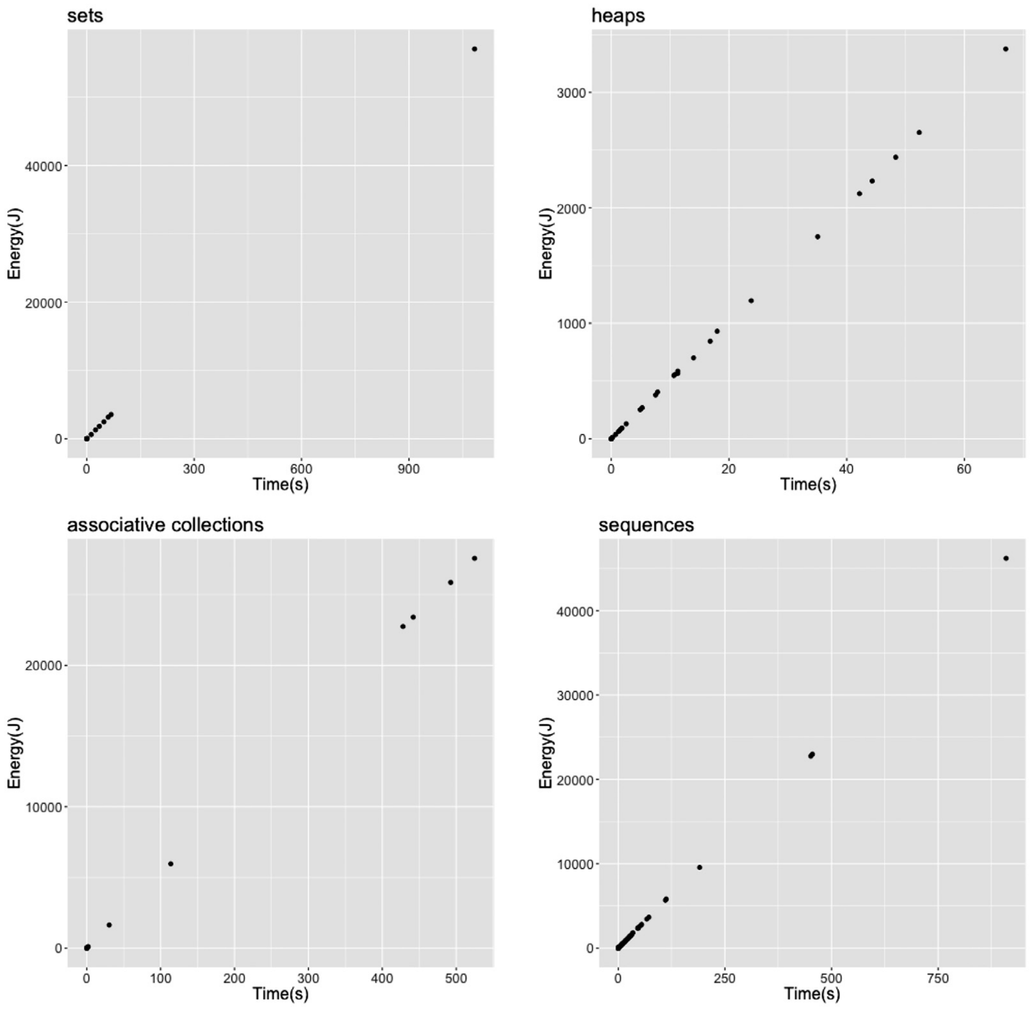


Fig. 1. Scatterplots for the relationship between energy (y-axis) and time (x-axis) within each abstraction considered.

tion: the StandardSet implementation consumes 29.4% of the time and 27.9% of the energy spent by UnbalancedSet.

For Sets, for all operations of the benchmark, the differences between the proportions of either time or energy consumption are always lower than 1.49%.

Heaps. As we have observed for Sets, our experiments suggest that energy consumption is proportional to execution time. Concrete evidence of this is shown in Figs. 3 (a) and (b), with the comparison between proportions of runtime and energy consumption for add and toArray, respectively, for each of the considered implementations.

Overall, the LazyPairingHeap implementation was observed to be the most efficient in all benchmark operations except for add. SkewHeap and SplayHeap implementations were the least efficient in 5 operations each. The proportions of runtime and energy consumption differ in at most 2.16% for any operation in any implementation of Heaps.

Associative Collections. Energy consumption was again proportional to execution time. The AssocList implementation was observed to be less efficient for all but the add and iterator operations. In the cases where AssocList was less efficient than StandardMap, the difference ranged from 9%, for addAll (depicted in Fig. 4 (a)), to 99.999% for retainAll. For the add and iterator operations, illustrated in Figs. 4 (b) and (c), StandardMap took approximately 40% and 85% more time and energy than AssocList. The proportion of consumed energy was (marginally, by 1%) higher than the proportion of execution time only for the add operation.

Sequences. The results obtained for Sequences also show that execution time strongly influences energy consumption. This is illustrated in Fig. 5 for the remove operation. The observed proportions across all operations and implementations differ at most in 1.9%, for the add operation.

4.5. In-depth analysis

The previously presented experiment provided us with a picture of the relative performance of the various implementations (for each abstraction) available in Edison, at a specific data point, or problem size. This, however, does not allow us to understand the variation of the time and energy consumptions with changing input sizes to the various benchmark operations considered.

In order to provide more information to help guide developers, we have designed and conducted an experiment in which we assess the time and energy consumption evolution, for a benchmark operation, with increasing base structure size (limited by the maximum size for each operation, for a specific data structure abstraction, as pictured in Tables 2 and 4).

In this additional experiment our strategy was to divide the maximum base size, for a specific operation, in five equidistant steps and run the benchmarks for each of those steps. The breakdown of the base size variation steps involved is presented in

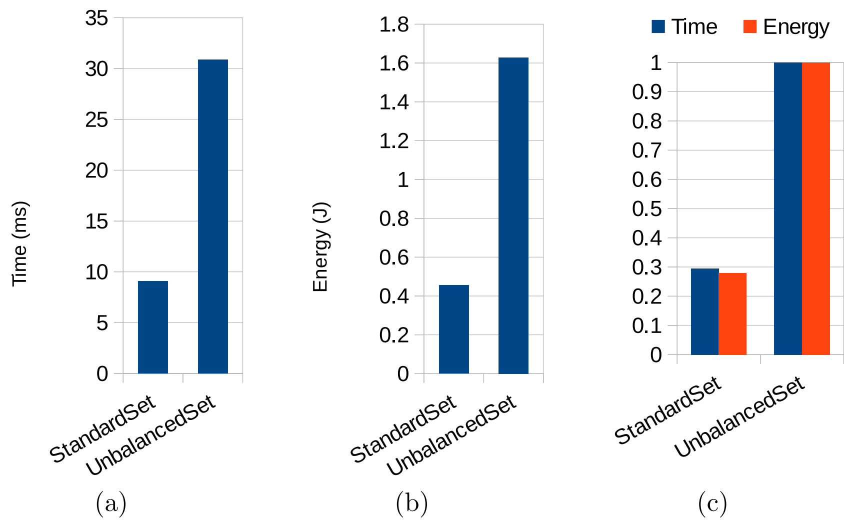


Fig. 2. Results of the `clear` operation for Sets.

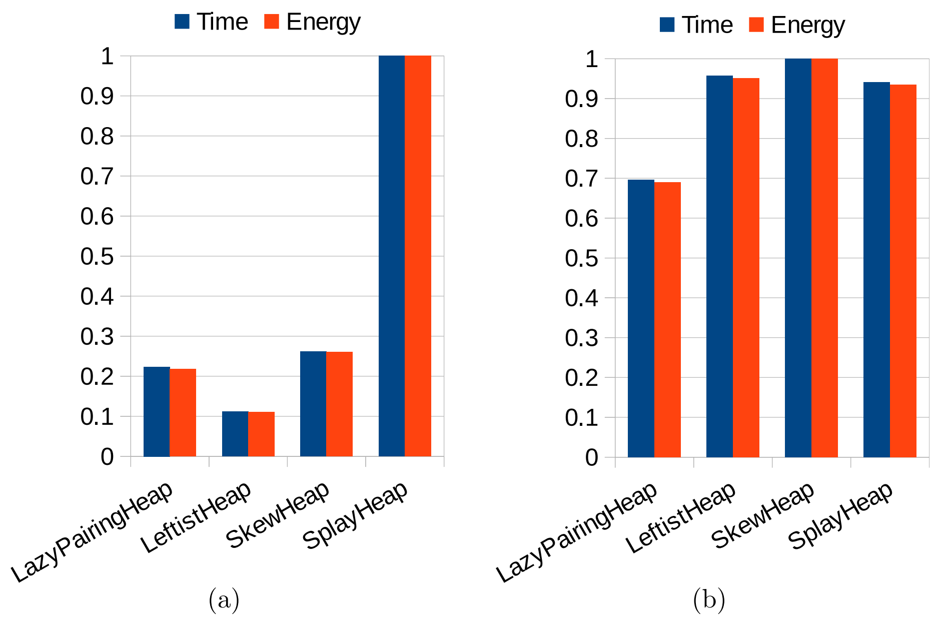


Fig. 3. Results of the `add` (a) and `toArray` (b) operations for Heaps .

Table 6
Base sizes for each abstraction and operation.

Abstraction	Operation	Bases
Sets	*all benchmark operations*	20000, 40000, 60000, 80000, 100,000
Heaps	*all benchmark operations*	20000, 40000, 60000, 80000, 100,000
	clear	10000, 20000, 30000, 40000, 50,000
	retainAll	5000, 10000, 15000, 20000, 25,000
	remove toArray	625, 1250, 1875, 2500, 3125
Associative Collections	*all other benchmark operations*	20000, 40000, 60000, 80000, 100,000
Sequences	add containsAll	625, 1250, 1875, 2500, 3125
	all other benchmark operations	20000, 40000, 60000, 80000, 100,000

Table 6. As an example, for the `Sequences` abstraction, for the `containsAll` operation, the maximum size of 3125 was divided into the steps 625, 1250, 1875, 2500 and 3125. For each of these values we obtained time and energy consumption measurements and then plotted their evolution. We did this, similarly, for each data structure abstraction (provided by `Edison`) and operation considered in the benchmark. The results of these experiments are presented next.

Overall, the observed tendency in the results was that:

i) with increasing base data structure size we see increased running time;
ii) longer execution times imply greater energy consumption.

Again, the correlation between execution time and energy consumption was statistically confirmed by the Spearman correlation coefficients and respective p-values given in Table 7.

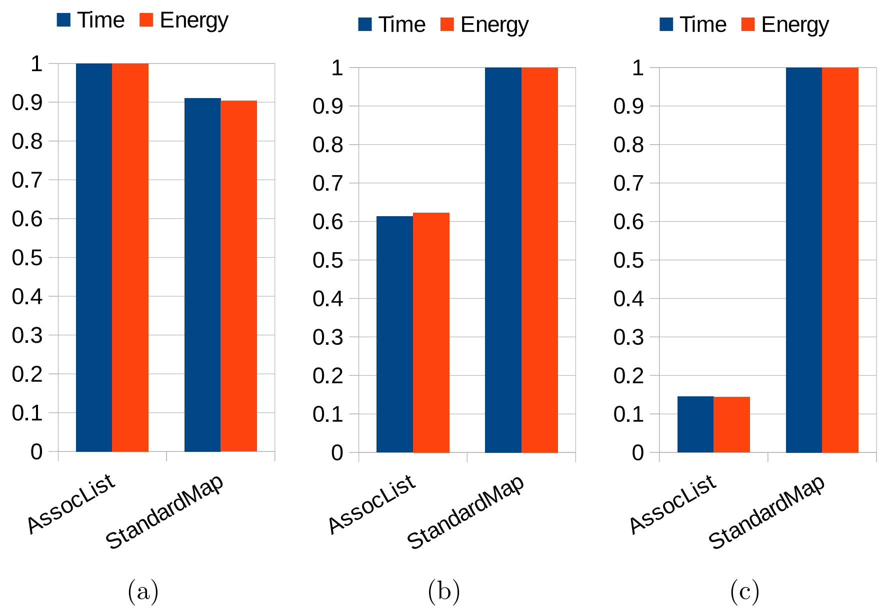


Fig. 4. Results of `addAll` (a), `add` (b), and `iterator` (c) for Associative Collections .

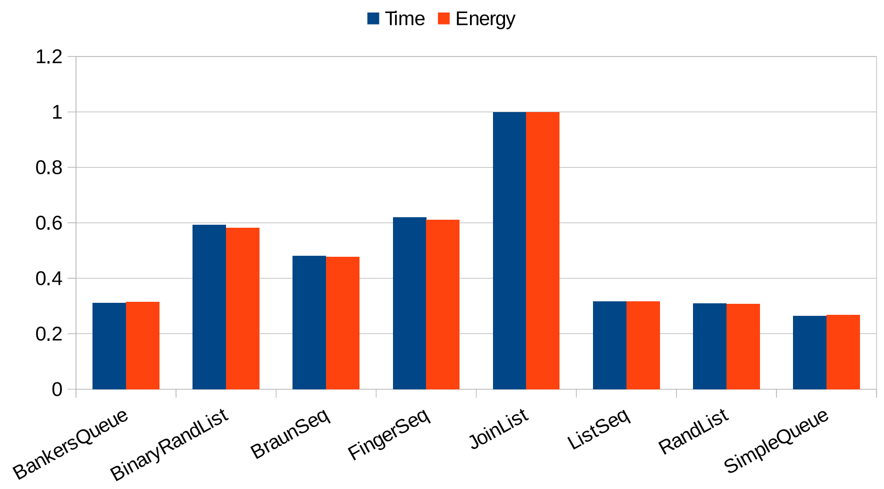


Fig. 5. Results of the `remove` operation for Sequences.

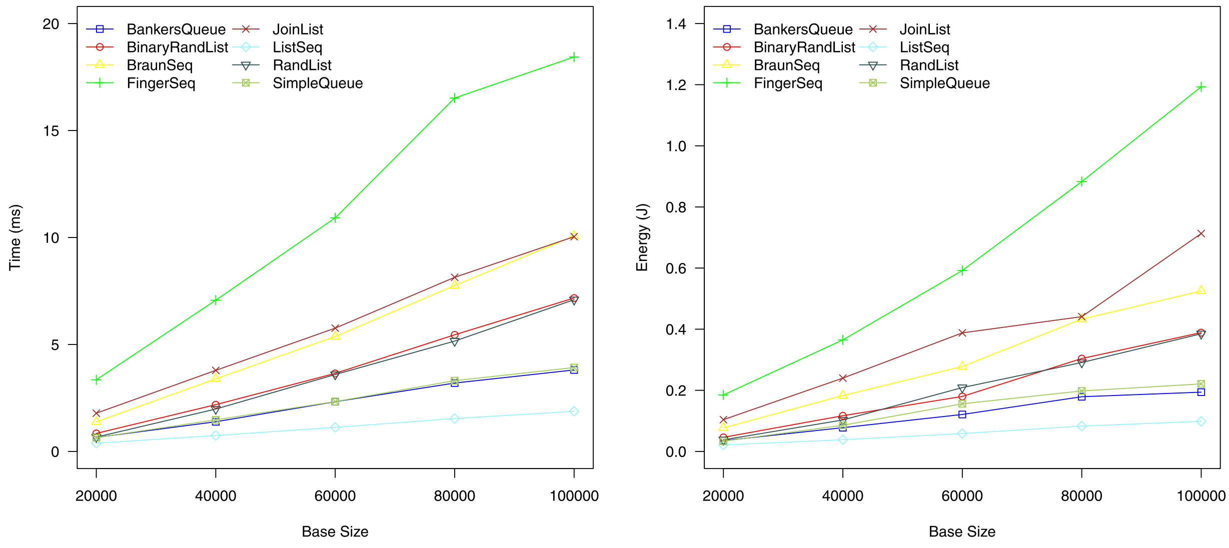


Fig. 6. Variability study for the iterator operation, for Sequences.

Table 7
Correlation between time and energy consumption for the analyzed abstractions.

Abstraction	Spearman correlation	p-value
Sets	0.999616	< 2.2e-16
Heaps	0.9986631	< 2.2e-16
Associative Collections	0.9991269	< 2.2e-16
Sequences	0.9991644	< 2.2e-16

Table 8
Abstractions and operations with diverging results.

Abstraction	Operations
Sets	add
Heaps	add
Associative Collections	add
	containsAll

This correlation is illustrated, for example, in Fig. 6 for the `iterator` operation, for the `Sequences` abstraction.

We can see in that figure that, for each (20000 elements) increment of the base data structure size, there is a corresponding (variable) increment of the execution time, in the top graph, and of the energy consumption, in the bottom graph, for every implementation of the `Sequences` abstraction. Nonetheless, there were a few diverging results. In Table 8 we present the abstractions and operations for which the obtained results deviated from the general case.

In those cases, the different behavior (to the most perceived pattern) we observed, can be, we concluded, attributed to the combination of our interpretation and implementation of the operations prescribed by the benchmark used, and the characteristics (properties) of operation (e.g. invariants maintained), of the various data structures provided by `Edison`.

For example, for the add operation, for the `StandardSet` implementation, as the number of elements in the base structure increases (to 100,000 elements), and because we insert the same elements already present in the base, the work performed in our experiment actually decreases.

A detailed account of the perceived reasons for the diverging results obtained in this analysis can be found in Appendix A, at the end of this work. The results of this work are available at the companion website, at green-haskell.github.io.

5. Comparing concurrent programming constructs

In this section, we present our second study, which aimed to assess the energy efficiency of `Haskell`'s concurrent programming constructs. This study is aims to address the following research questions:

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?

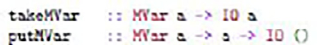
We start out by briefly presenting the concurrent programming primitives we analyzed in this study (Section 5.1). Section 5.2 then describes the set of benchmarks that we used. Since we analyzed multiple variants of each benchmark, Section 5.3 explains how we adapted them to produce these variants. Finally, Section 5.4 presents the results of the study.

5.1. Concurrency in `Haskell`

Non-deterministic behavior in `Haskell` can be implemented through the use of the `IO` type. With it, we describe sequences of operations that can perform *IO* and emulate mutable behavior just like in any imperative programming language. The `IO` type also encapsulates computations that are performed in different threads. To create a new thread, we can choose between three different functions: `forkIO`, which creates a new lightweight thread to be managed by the language's scheduler; `forkOn`, which creates a lightweight thread to be executed on a specific processor; and `forkOS`, which creates a thread bound to the OS's thread structure. More specifically, `forkOn` attempts to perform thread pinning by establishing that a thread has strong affinity with a given physical core, whereas `forkOS` establishes that a Haskell thread is always associated with the same OS thread. The latter is useful for applications that interface with code written in other languages that require thread-local state.

The number of `Haskell` threads that can run truly simultaneously at any given time is determined by the number of available *capabilities*. Capabilities are virtual processors managed by the `Haskell` runtime system. Each capability can run one `Haskell` thread at a time. A capability is animated by one or more OS threads. It is possible to configure the number of capabilities `N` used by the runtime system.

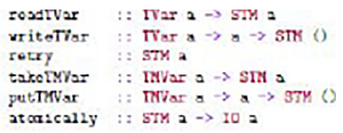
The basic data sharing primitive of `Haskell` is `MVar`. An `MVar` can be thought of as a box that is either empty or full. `Haskell` provides two main operations to work with `MVars`:



```
takeMVar    :: MVar a -> IO a
putMVar     :: MVar a -> a -> IO ()
```

Function `takeMVar` attempts to take a value from an `MVar`, returning it wrapped in a value of type `IO`. The operation succeeds for full `MVars`, but blocks for empty ones until they are filled. Conversely, `putMVar` attempts to put a value into an `MVar`. The operation succeeds for empty `MVars`, and blocks for full ones until they are emptied. `MVars` combine locking and condition-based synchronization in a single abstraction.

The implementation of software transactional memory for `Haskell`, called STM Haskell (Harris et al., 2005), provides the `TVar` type to implement mutable variables that only transactions manipulate. The type signatures for STM functions are the following:



```
readTVar    :: TVar a -> STM a
writeTVar   :: TVar a -> a -> STM ()
retry       :: STM a
takeTMVar   :: TMVar a -> STM a
putTMVar    :: TMVar a -> a -> STM ()
atomically  :: STM a -> IO a
```

Functions `readTVar` and `writeTVar` return and modify the value of `TVars`, respectively, and return a value of type `STM`. To run as a transaction, an `STM` value is executed by the `atomically` function. Building upon the definition of `TVars`, STM Haskell also provides another type, `TMVar`, and corresponding operations for taking values from and putting values into a `TMVar`. As the name implies, it is a transactional variant of the `MVar` type. Operations on values of type `TMVar` produce values of type `STM` as result.

5.2. Benchmarks

We selected a variety of concurrent `Haskell` programs to use as benchmarks in our study. Benchmarks `chameneos-redux`, `fasta`, `k-nucleotide`, `mandelbrot`, `regex-dna`, and `spectral-norm` are from The Computer Language Bench-

Table 9
The benchmarks employed in this study.

Benchmark	Description
`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.
`fasta`	This benchmark generates random DNA sequences and writes it in FASTA format[a]. 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.
`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.
`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 iterated on it. The only synchronization point is the main thread waiting for the result of each worker.
`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.
`spectral-norm`	The spectral norm is the maximum singular value of a matrix. Synchronizes workers using a cyclic barrier.
`dining-philosophers`	An implementation of the classical concurrent programming problem posed by Dijkstra. The philosophers perform no work besides manipulating the forks and printing a message when eating.
`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 and Adl-Tabatabai, 2011).
`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, part of the Yesod Web Framework. This benchmark was inspired by the Tomcat benchmark from DaCapo (Blackburn et al., 2006).

[a] https://en.wikipedia.org/wiki/FASTA_format.

marks Game[3] (CLBG). CLBG is a benchmark suite aiming to compare the performance of various programming languages. Benchmark `dining-philosophers` is from Rosetta Code,[4] a code repository of solutions to common programming tasks. Benchmarks `tsearch` and `warp` were developed by us. The latter is based on the Tomcat benchmark of the DaCapo (Blackburn et al., 2006) benchmark suite. Table 9 presents descriptions for all the benchmarks.

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.

5.3. Methodology

In order to use the suite of benchmarks described in the previous section 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. As a result, each benchmark has up to 9 distinct variants covering a number of different combinations. 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 used also as a condition-based synchronization mechanism. In such cases, we did not create `TVar` variants as `TMVar` mimics exactly this behavior, while using `Haskell`'s STM. In other benchmarks like `tsearch` and `warp` we changed only the thread management construct as they are complex applications and it wouldn't be straightforward to change the synchronization primitives without introducing potencial bugs.

Each variant we created is a standalone executable. This executable is a `Criterion` microbenchmark that performs the experiment by calling the original program entry point multiple times. We run this executable 9 times, each one changing the number N of capabilities used by the runtime system. We used the following values for N: {1, 2, 4, 8, 16, 20, 32, 40, 64}. The numbers 20 and 40 correspond to the number of physical and virtual cores of the machine on which we ran the experiments. In the analysis of the performance and energy efficiency of parallel systems and benchmarks, it is commonplace to use powers of two for the number of threads/processes/cores/units of computation (e.g, Pinto et al. (2014); Ribic and Liu (2016)) since differences are usually not significant when considering small increments in the number of these units. An alternative approach is to vary the number of units of computation linearly with a coefficient larger than one (e.g., David et al. (2013) uses multiples of 6 and 10).

The **independent variables** in our study are number of capabilities (9), thread management construct (3), concurrency control primitive (3), and benchmark (9). The overall number of configurations is approximately 500 since not every possible combination is valid, e.g., `tsearch` does not leverage any concurrency control primitive. The **dependent variables** are execution time and energy consumption.

5.4. Results

In this section, we report the results of our experiments with concurrent `Haskell` programs. The results are presented in Fig. 8. Here, the odd rows are energy consumption results, while the even rows are the corresponding running time results. Navigable versions of these graphs 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

[3] http://benchmarksgame.alioth.debian.org/.
[4] http://rosettacode.org/.

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 6.

Faster is not always greener. Overall, the shapes of the curves in Fig. 8 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 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 5.5.

To complement this analysis, we have also calculated the Spearman correlation between time and energy consumption. This is a non-parametric correlation coefficient. To perform this calculation, for each benchmark, we used as data points the results for time and energy consumption while considering every experimental configuration for which we executed said benchmark. The number of (time, energy) pairs of data points varied from 27 for warp and tsearch (three thread management constructs and nine numbers of capabilities) to 81 for fasta (three thread management constructs, three concurrency control primitives, and nine numbers of capabilities). The correlation between time and energy consumption was in general very high. The highest was 0.985 for dining-philosophers and warp. Nonetheless, there was much more variation than in the data structures study; for fasta the correlation was 0.545, for tsearch it was 0.643, and for regex-dna it was 0.727. These results suggest a strong but imperfect relationship between time and energy consumption. Table 10 presents the obtained correlations for all the benchmarks. Fig. 7 graphically depicts the relationship between energy and time for the nine benchmarks using scatterplots.

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.

Table 10
Correlation between time and energy consumption for the analyzed benchmarks.

Benchmark	Spearman correlation	p-value
chameneos-redux	0.980	$< 2.2e-16$
fasta	0.545	1.441e−07
dining-philosophers	0.985	$< 2.2e-16$
k-nucleotide	0.916	$< 2.2e-16$
mandelbrot	0.927	$< 2.2e-16$
regex-dna	0.727	$< 2.2e-16$
spectral-norm	0.961	$< 2.2e-16$
tsearch	0.643	0.0003996
warp	0.985	1.039e−06

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 Section 7.

5.5. In-depth analysis: *fasta*

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

5.5.1. 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.

5.5.2. The fastest consume more energy

In Fig. 8, 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

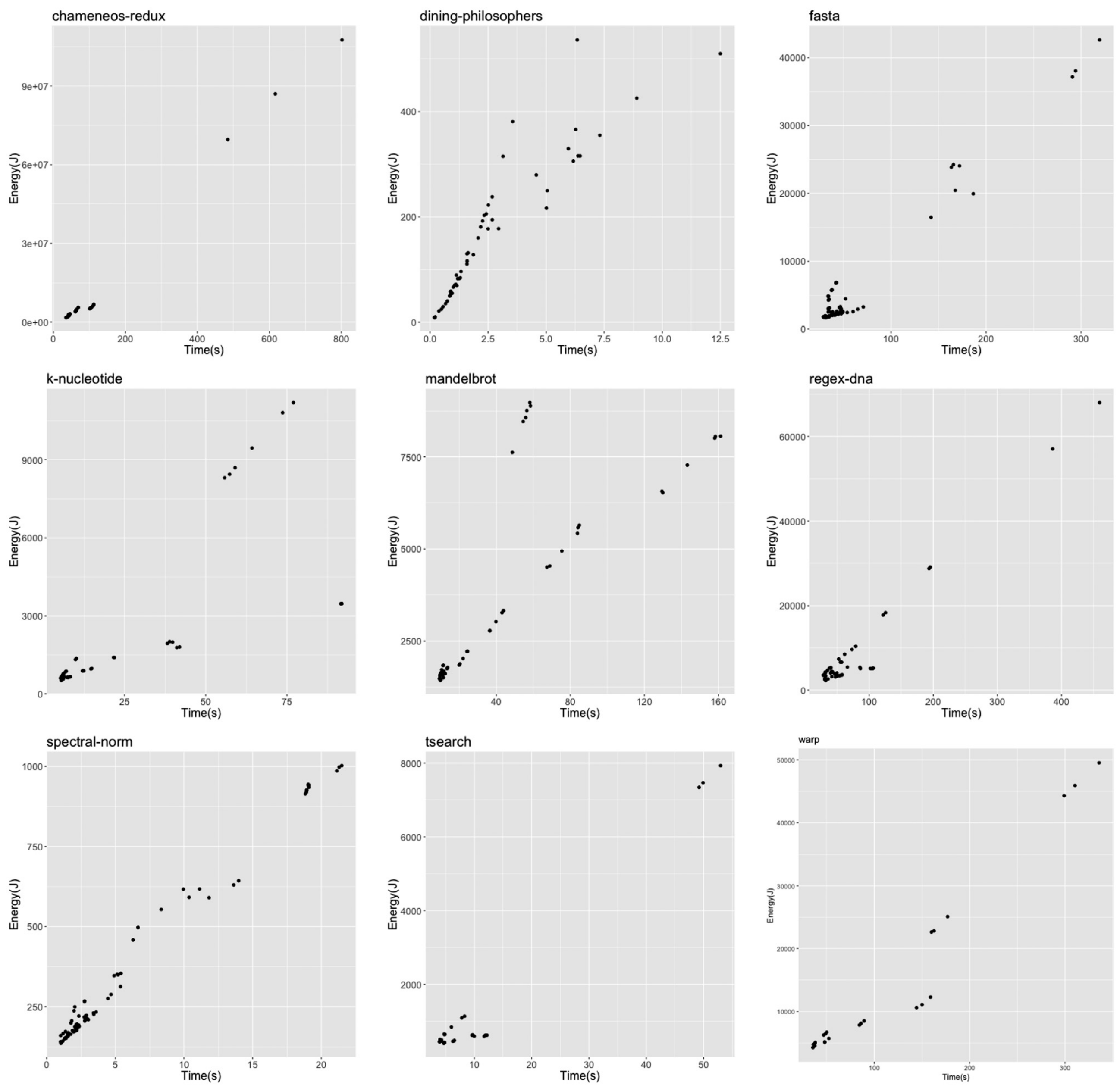


Fig. 7. Scatterplots for the benchmarks for the relationship between energy (y-axis) and time (x-axis).

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`[5] 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 et al., 2009) readings of the log can be seen in Figs. 9 and 10 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.

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, 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 5.5.1) into the shared variable, the other transactions are aborted and retry.

5 http://ghc.haskell.org/trac/ghc/wiki/EventLog.

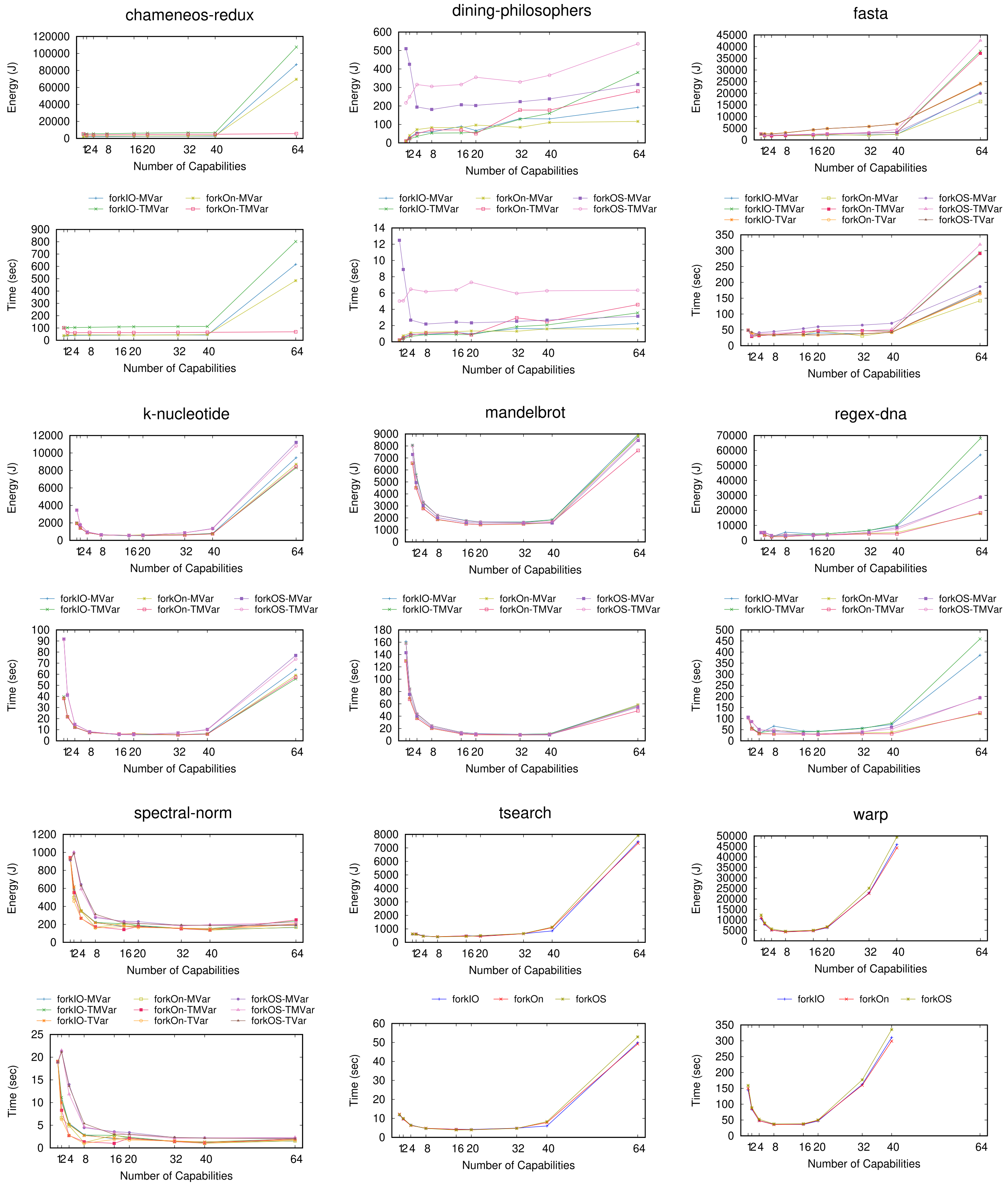


Fig. 8. Energy/Time with alternative concurrency abstractions and varying the number of capabilities.

Considering this scenario we just described, a possible explanation 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`[6] 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. Fig. 11 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 suc-

6 http://hackage.haskell.org/package/stm-stats.

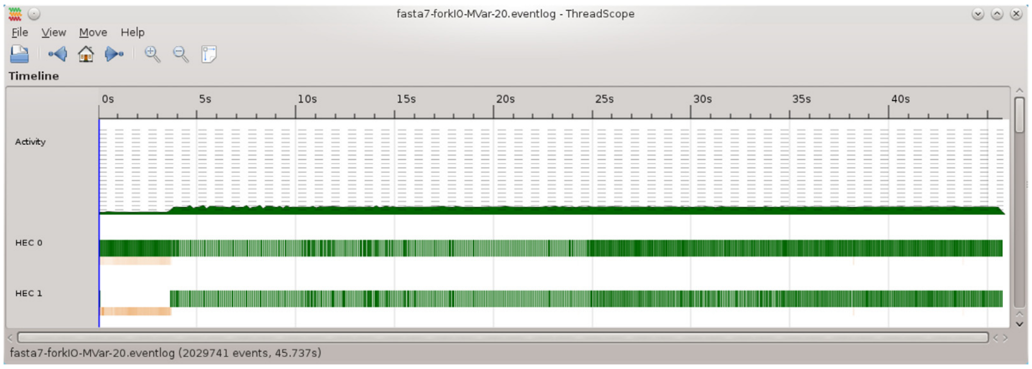

fasta7-forkIO-MVar-20.eventlog - ThreadScope
File View Move Help
Timeline
0s
5s
10s
15s
20s
25s
30s
35s
40s
Activity
HEC 0
HEC 1
fasta7-forkIO-MVar-20.eventlog (2029741 events, 45.737s)

Fig. 9. ThreadScope readings for the `forkIO-MVar` variant of `fasta`.

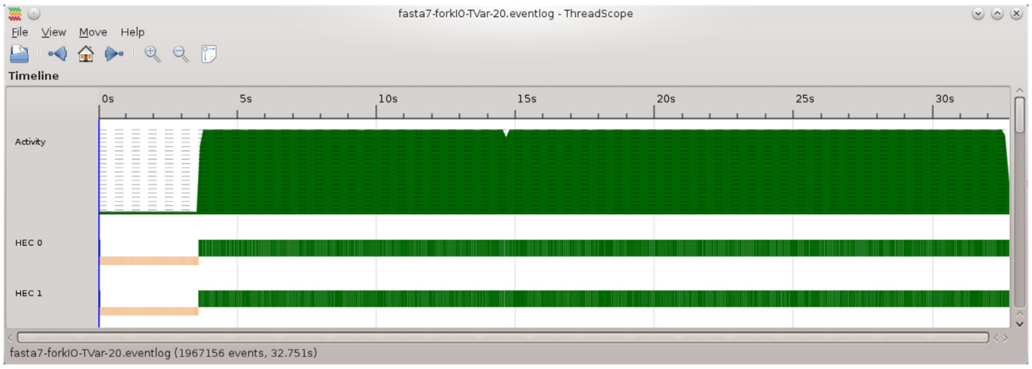

fasta7-forkIO-TVar-20.eventlog - ThreadScope
File View Move Help
Timeline
0s
5s
10s
15s
20s
25s
30s
Activity
HEC 0
HEC 1
fasta7-forkIO-TVar-20.eventlog (1967156 events, 32.751s)

Fig. 10. ThreadScope readings for the `forkIO-TVar` variant of `fasta`.

```
STM transaction statistics (2016-07-20 19:16:02 UTC):
Transaction        Commits    Retries      Ratio
generate-numbers       299       4138      13.84
output-sync            261         33       0.13
wait-semaphore           2          2       1.00
```

Fig. 11. Output of `stm-stats` for `forkIO-TVar` variant of `fasta`.

ceeded while 4138 others failed, which represents 13.84x more executions than necessary.

6. Discussion

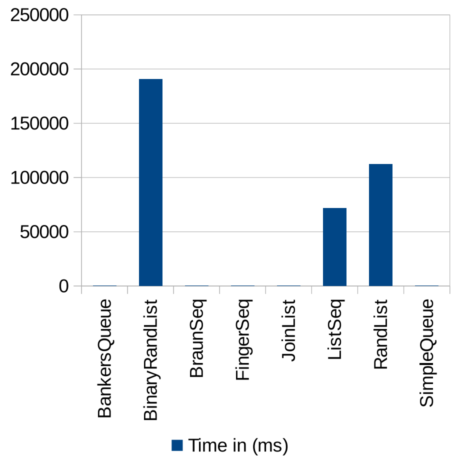
Generally, especially in the sequential benchmarks, high performance is a proxy for low energy consumption. Our first study (Section 4) highlighted this for a number of different data structure implementations and operations. Concurrency makes the relationship between performance and energy less obvious, however. 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 constructs 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 7 benchmarks where we implemented variants using different data sharing primitives, in 5 of them the thread management construct had a stronger impact on energy usage than the data sharing primitives. Furthermore, in these 5 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 best `forkIO` variant consumes around 15% more than the best `forkOn` variant (1653 vs. 1432 J). In `spectral-norm`, the best `forkOS` variant consumes 34% more energy than the best `forkOn` variant (183 vs. 136 J). 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.

Although there is no overall winner, for most benchmark we can point out a configuration that beats the others in terms of energy consumption and performance. We can observe that the ordering of the curves is more or less preserved when comparing the graph of each metric. This is useful for developers because: (1) small changes can make big differences; (2) it is very easy to



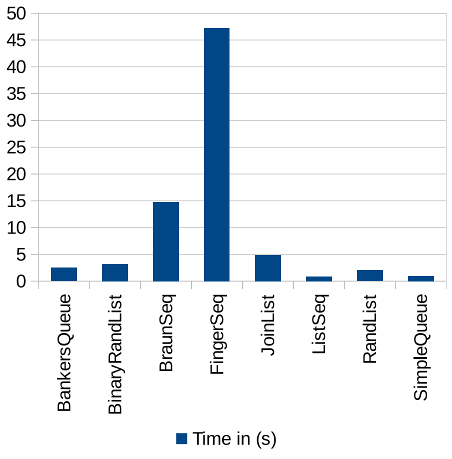


Fig. 12. Time results of the `add` (a), and `contains` (b), operations for Sequences.

change concurrent constructs; (3) it is cheap to experiment and perform benchmarks; and (4) it is easy to identify which configuration excels.

7. Guidelines for `Haskell` developers

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

It is important to point out that these guidelines are based solely on our experimental results from Sections 4 and 5. 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 their programs 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 similar results in other scenarios.

7.1. Performant data structures are greener

Description: The energy consumption of a sequential program seems to follow proportionally from its time consumption; so, when selecting data structure implementations, it is recommendable to select the ones with greater performance on the operations that are stressed out the most. Stated another way, the Algorithmic Complexity of a particular operation, for a specific data structure implementation is a solid indicator of its energy consumption performance, at least in sequential scenarios.

Rationale: Data structure libraries such as Edison provide different implementations of the same abstractions. This is particularly convenient for developers: whenever a program needs to rely on a concrete abstraction, different *certified* implementations for it are readily available, together with standard functions on it. This is precisely what we have explored: given the benchmark operations that we wanted to implement, we relied on concrete Edison functions. We then analyzed the performance of such operations, and we were able to provide detailed and reusable information on them. Concretely, this information can be of value, e.g., as follows: if a program needs to rely on a `Sequence` essentially for using operations `add` and `contains`, the implementation to select is `SimpleQueue`, as can be observed in Fig. 12.

Generally speaking, when choosing a concrete implementation of a data structure to use in a program, it is recommended to opt for the one that is faster on the operations the program requires.

7.2. 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 and Shavit, 2012). Three of the benchmarks from Section 5 fit this description: `mandelbrot`, `regex-dna`, and `spectral-norm`. The results from our study have shown that, for these benchmarks, the variants using `forkOn` superseded the others in both performance and energy consumption. These results show that manually distributing the workload in an even manner among the capabilities instead of handing this job to the runtime system scheduler improves performance. This makes sense because we know beforehand that each worker thread is doing a similar amount of work. In such scenarios, there is no need to migrate a thread from one capability to another since 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 Section 5, 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 Fig. 13, 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.

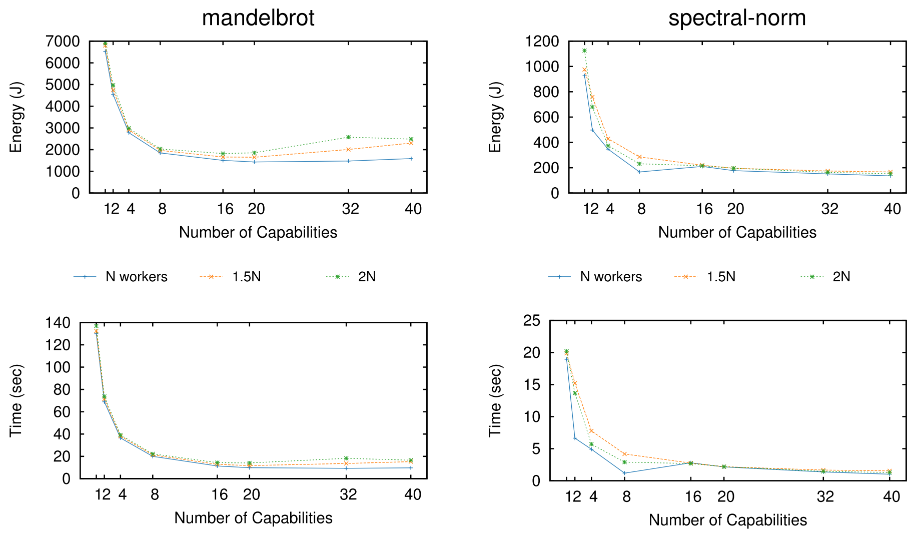


Fig. 13. Performance of `mandelbrot` and `spectral-norm` with different number of workers.

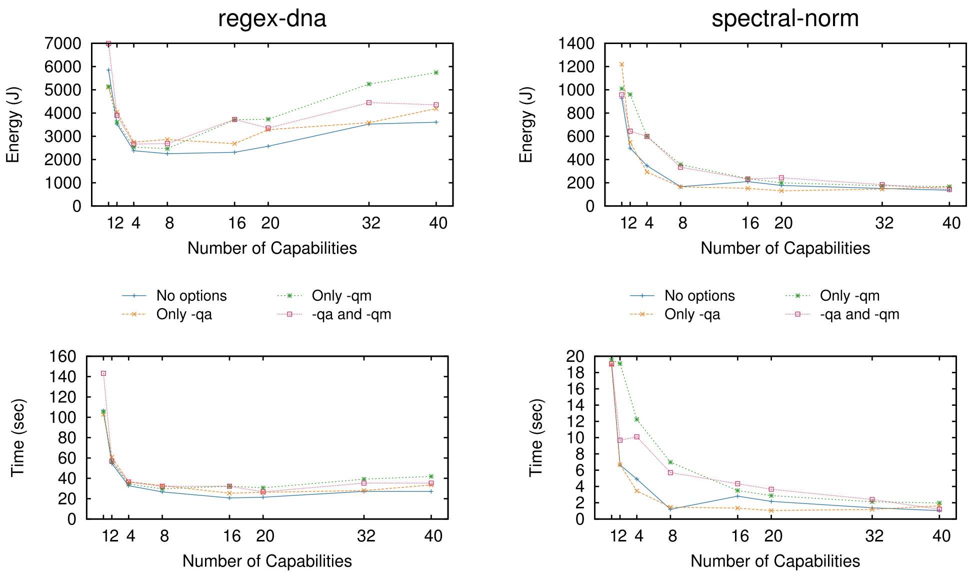


Fig. 14. Performance of `regex-dna` and `spectral-norm` using `-qa` and `-qm`.

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[7]. Using this option, the OS threads associated with a capability *i* are bound to 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 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 Fig. 14 we show the 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 that developers experiment with these options to assess how they affect the performance of a given program.

7.3. In CPU-bound applications, avoid setting more capabilities than available CPUs

Description: Using more capabilities than the number of available virtual CPU cores in CPU-bound applications 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

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

from Section 5, 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. This is clearly observable in Fig. 8.

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."*.[8] 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.

7.4. *Avoid using `forkOS` to spawn new threads*

Description: Using `forkOS` inadvertently 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 Section 5, 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 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.

7.5. *Only use STM if transaction conflicts are rare*

Description: Transactions can improve performance in scenarios where synchronization would need to be employed intensively. However, this may come at the cost of many aborted transactions and, even though these transactions do not hinder performance, they impose a heavy energy penalty.

Rationale: In scenarios where synchronization is employed intensively, an optimistic approach such as transactional memory can improve the performance of a program. In the `fasta` benchmark, versions of the benchmark using `TVar` exhibited very competitive performance, outperforming `MVar` versions in most configurations. However, this came at a cost: versions of the benchmark that employed transactional memory (both `TMVar` and `TVar`) exhibit the most intense energy consumption. As explained in Section 5.5, the problem was that thousands of transactions were being aborted, which meant that multiple threads were performing useless work and this imposed a burden on the `Haskell` runtime system.

This situation is to be expected when opportunities for parallelization are scarce due to dependencies between the threads. In `fasta`, the greatest improvement that could be achieved over a sequential version was approximately 37.5%. This is not negligible, but it is small when we consider that the experiments were run on a 20 core server. Hence, if energy consumption is important in a `Haskell` program, developers should carefully profile the program before committing to the use of transactions. In this manner, they can avoid scenarios where many transactions are aborted.

8. 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 the data structures available in the Edison library and a subset of `Haskell`'s constructs for concurrent and parallel programming. It is not possible to extrapolate the results to other data structure implementations or to alternative 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 the two studies to other hardware platforms for which `Haskell` programs can be compiled. Factors such as operating system scheduling policies (Yuan and 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. These additional experiments targeted both studies.

For the first study, the conclusions we have drawn in the paper are consistent with the results we obtained on one different machine, a 4-core Intel Core i7-4790 (Haswell) with 16 GB of DDR 1600 running openSUSE 13.2 and GHC 7.10.2. For the second study, we ran some of the benchmarks on another machine, a 4-core Intel i7-3770 (IvyBridge) with 8 GB of DDR 1600 running Ubuntu Server 14.04.3 LTS (kernel 3.19.0–25) and GHC 7.10.2. Fig. 15 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 Fig. 8. The same trend can be observed for the remaining benchmarks.

In this work, we have collected energy consumption data from the PKG domain only. While it is not possible to generalize our results considering all the domains for which RAPL provides energy data, analyzing the PKG domain is of primary importance. Also, recent work (Melfe et al., 2018) has considered our first study, in its original version of Lima et al. (2016), but also incorporating data from the DRAM domain, and the results it describes are completely aligned with our own.

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.

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

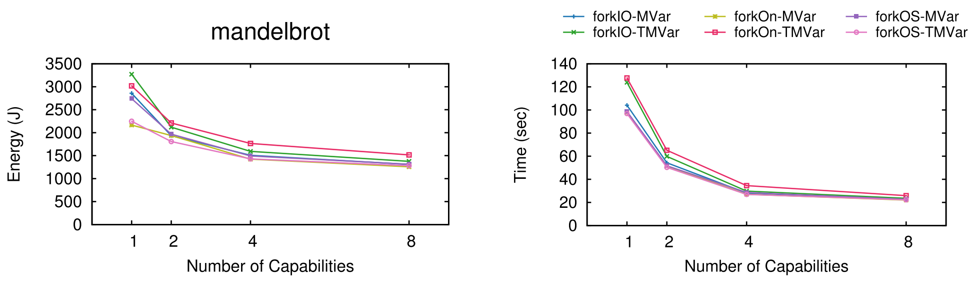


Fig. 15. Energy/time on alternative platform.

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 behavior, it can have a significant impact on performance, especially for concurrency. For strictness, the work of Melfe et al. (2018) considered the different GHC compiler optimization options and also showed that the faster the compiled code is the more energy efficient it is. 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.

9. Related work

Murphy-Hill et al. (2009) provide an analysis on the use of refactoring. Their study indicates how refactoring is common, even if only executed manually. Dig et al. (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 et al. (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 et al. (2012) specified and implemented refactorings for introducing parallelism into `Haskell` programs, considering mainly performance concerns. 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.

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 et al. (2014) 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 et al. (2014a) provide an analysis of the effects of code refactorings on energy consumption for 9 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. Two recent papers (Oliveira et al., 2017; Pereira et al., 2017) have studied the impact of using different programming languages on energy. Pereira et al. (2017) compared the energy behavior, performance, and memory consumption of 10 benchmarks across 27 different programming languages. They found out that the C versions of the benchmarks tended to exhibit the best performance and energy efficiency, but not always. Oliveira et al. (2017) studied the energy efficiency of more than 30 benchmarks and 4 apps running on Android, comparing C++, JavaScript, and Java versions. Although there was no overall winner, JavaScript versions tended to consume less energy. Our paper focuses on `Haskell` programs and the impact of changes regarding concurrent structures used and strictness of evaluation. 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. Scanniello et al. (2015) studied the migration of a performance-intensive system to an architecture based on GPU as a way to reduce energy waste. 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. Bruce et al. (2015) used Genetic Improvement to reduce the energy consumption of applications, reaching up to 25% reduction. All these approaches show the potential for program transformation, in general, and refactorings, in particular, to reduce energy consumption. We explore this potential further in this paper by targeting `Haskell`.

10. Conclusions

As energy efficiency becomes a popular concern for software developers, we must be aware of the implications of our development decisions in our applications energy footprint. In this paper, we analyzed a relevant subset of those decisions for a purely functional programming language, `Haskell`. We found that for sequential `Haskell` programs, execution time can be a good proxy for energy consumption. However, when considering concurrency, we found no silver bullet. In one scenario, choosing `MVars` over `TMVars` can save 60% in energy, while in another, `TMVars` can yield up to 30% energy savings over `MVars`, and performance is not always a good indicator. We have extended two tools for helping developers test their programs energy footprint: the `Criterion` benchmarking library, and the profiler that comes with GHC.

Acknowledgements

This research was partially funded by INES 2.0, (FACEPE PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, and CNPq

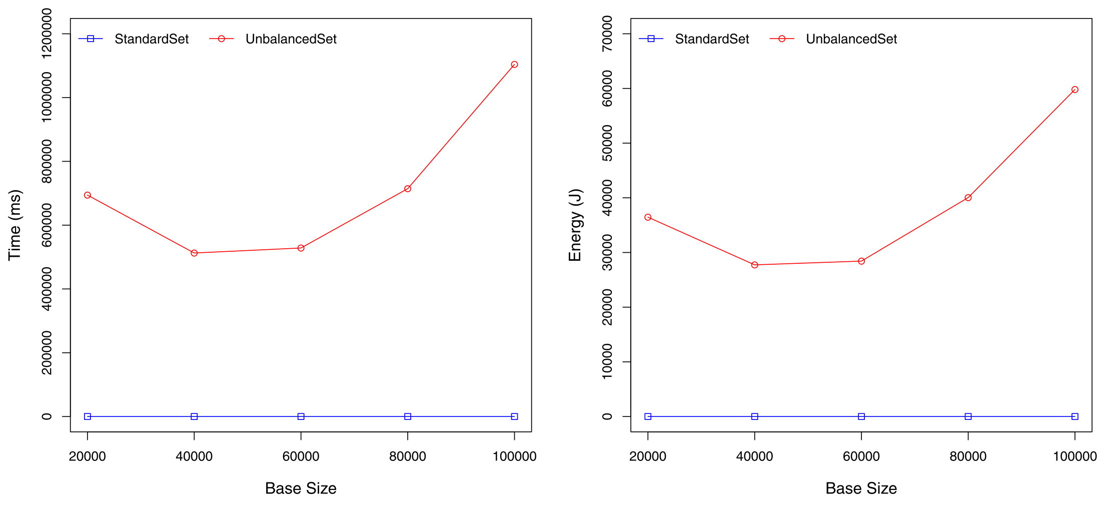


Fig. A16. Variability study for the add operation, for Sets.

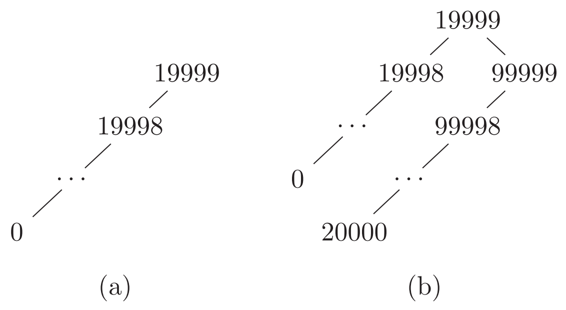


Fig. A17. (a) Base data structure (UnbalancedSet) with 20,000 elements (b) Data structure (UnbalancedSet) with 100,000 elements inserted atop a base with 20,000 elements.

465614/2014-0), FACEPE APQ-0839-1.03/14, and CNPq 304220/2017-5. It has also been financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718.

Appendix A. Appendix

In this appendix, we elaborate on the unexpected results obtained, in the analysis of the time and energy consumption evolution, with increasing base structure size, for the data structures in-depth analysis (Section 4.5). As discussed in that section, the overall tendency observed, was that:

i) with increasing base data structure size we see increased running time;
ii) longer execution times imply greater energy consumption.

In Sections A.1 and A.2, we discuss results that do not conform with (i) and (ii), respectively.

A1. Runtime is not proportional to base structure size

Sets

For `Sets`, the `add` operation presents us with a non-obvious behavior pictured in Fig. A.16.

In order to analyze this behavior we need to explain our design choice for creating the base data structures and how the elements to be inserted into that data structure are actually inserted.[9]

The base data structure creation relies on the insertion function present in the benchmark, the `add` operation. To create a base data structure of size n, we insert, in this order, the values {n-1, n-2, ., 0} into an empty data structure. Then, irrespective of the base size, 100,000 elements will be added to the base data structure, resorting to the `add` operation. This is achieved by inserting, in this order, the values {99999, 99998, ., 0} into the base data structure.

For a base data structure of size 20000, this means that we start by inserting 80,000 elements that are not in the base data structure, and then insert 20,000 that are already there.

For the other iterations of the experiment the same method is followed, varying only the number of elements in the base; e.g. for the last iteration we insert 100,000 elements, into a base data structure already containing that same number of elements, in fact, the same elements, which were inserted in the same (decreasing) order.

Now we can proceed to explain some of the unexpected observed behaviors.

For the `UnbalancedSet` implementation, we can see that, the execution time measurements, collected as the size of the base data structure increases, form a concave upward curve.

This can be explained thus: the `UnbalancedSet` implementation is realized as an unbalanced binary search tree in which duplicates are not allowed; also, we use the `add` operation to insert the elements into the base data structure, as well as for actually doing the insertions prescribed by the benchmark.

For the first iteration of the experiment a base data structure with 20,000 elements is created; taking into account the insertion method described previously, this will result in a degenerate tree with 20,000 elements hanging to the left, as pictured in Fig. A.17(a).

Then 100,000 elements will be inserted into that data structure; the first 80,000 of which, are not yet in the base data structure and, because they are bigger, will be inserted to the right of the tree's root, resulting in the tree pictured in Fig. A.17(b); this tree has 100,000 elements and a height[10] of 80000.

In the second iteration the base data structure's size is 40000; the resulting tree, although having the same 100,000 elements (be-

[9] This process was explained here, in the `Sets` context, but is actually the same for the other abstractions.

[10] The length of the longest path from the root of the tree to a leaf.

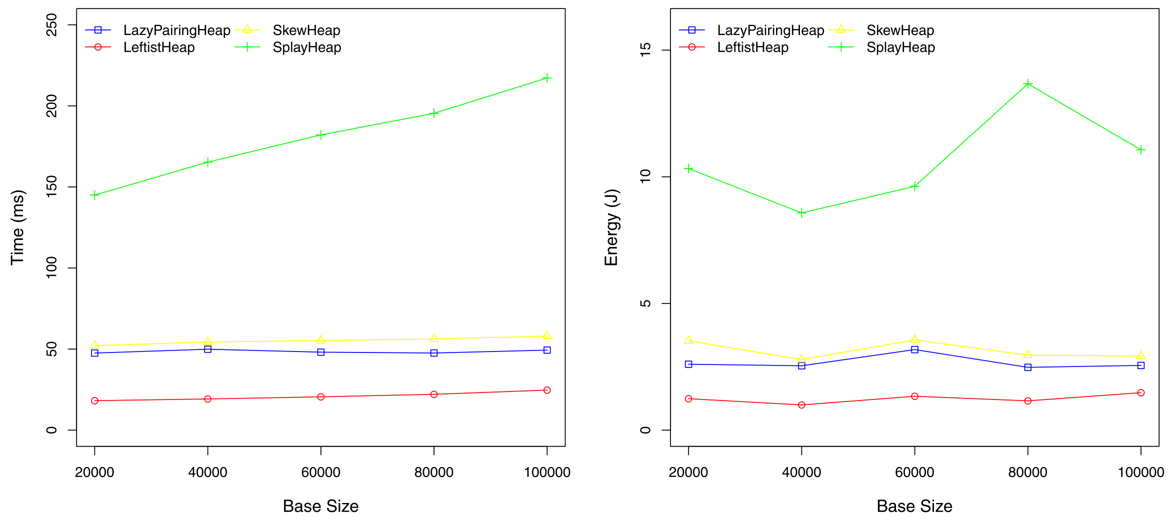


Fig. A18. Variability study for the add operation, for Heaps.

ing “divided” in a “base” left branch with 40,000 elements and a right branch with 60000), will have a height of “only” 60000; this decrease in height (which is a determinant factor for the performance of operations over a tree when elements end up being inserted on its leaves) accounts for the reduction in time consumption.

The two following iterations are dual to the first two, reversing the left (base)/right branches, and the decreasing tendency; in the final iteration the tree has degenerated into a 100,000 elements “list” (there is only a left branch, therefore its height is 100000) which explains the final increase in time consumption.

For the `StandardSet` implementation, we can see that, the execution time is extremely small when compared with that of the `UnbalancedSet` implementation; it might not be perceivable from the graph, but the values for time are decreasing as the base size is increasing.

This behavior can be explained as follows: in the first iteration of the experiment a base data structure of size 20,000 is created, and 100,000 elements are inserted into it. Of those 100,000 the first 80,000 are elements which are not present in the base data structure and the remaining 20,000 are already present.

If we look at the `StandardSet` implementation we see that it is a “wrapper” around the Data.Set standard `Haskell` library, which is itself based on a binary search (without duplicates), size balanced, tree.

Seeing that, as we insert elements, the tree has to be (re)balanced, one might expect that the work involved would increase.

But the fact is, that the number of insertions for which the rebalancing has to be made is decreasing; in the first step 80,000 insertions trigger the rebalancing, in the second step this number decreases to 60000, until the last step, in which all of the 100,000 elements inserted already exist in the base data structure and therefore no balancing of the (sub)trees occurs.

Heaps

For the `add` operation, for the implementations other than `SplayHeap`, the results, shown in Fig. A.18, differed from the norm.

Indeed, for those implementations, the execution time varies very little throughout the range of base data structure size increments.

For the `SkewHeap` implementation this can be explained thus: a base data structure is created in the way already explained. For this `Heap` implementation the underlying data structure is a bi-

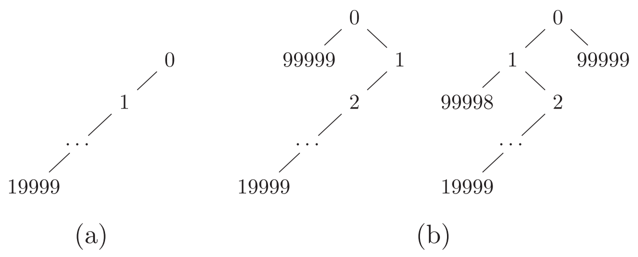


Fig. A19. (a) Base data structure (SkewHeap) with 20,000 elements (b) Data structure (SkewHeap) with the elements 99,999 and 99,998 inserted into a base with 20,000 elements.

nary tree, and our insertion process results in a completely unbalanced tree. For example, for a 20,000 elements base, the insertion process results in the tree depicted in Fig. A.19 (a). This structure, as well as all other SkewHeaps, respects the `Heap` order invariant: each node “is smaller” than its children.

The insertion of the 100,000 elements (as prescribed by the benchmark) into each base data structure follows the same approach: elements are inserted in decreasing order, from the maximum (99999) to the minimum (0).

During that insertion the implementation’s algorithm pushes entire trees/subtrees downwards if trying to insert an element smaller than, or equal to, the one at the root. If trying to insert a larger element it recursively inserts that element into the right subtree, and swaps the left and right subtrees; an example is pictured in Fig. A.19 (b) where the element 99,999 has been inserted into the base data structure with 20,000 elements, and the element 99,998 has been inserted into the resulting tree.

In the end, what happens is that, since we are adding in decreasing order (matching the heap order invariant) and because the implementation swaps branches, effectively alternating insertions between the left and right branches, the insertions are actually happening at, or close to, the root of the tree.

Therefore, performance-wise, we are actually measuring the exact same (insertion) effort. Indeed, for any base structure, the resulting tree is actually exactly the same except for one single (degenerated left) branch that was originated in the creation of the base structure. This branch is proportionally taller as the base structure size increases. Of course, in each case, we end up with an increasingly larger tree, but the difference occurs in the effort it takes to build the base data structures, which is beyond what

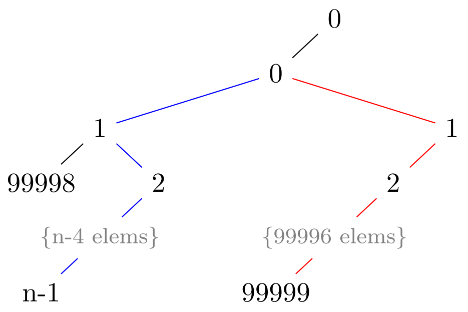


Fig. A20. Data structure (LeftistHeap) with 100,000 elements inserted into a base with n elements. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

we measure. So, the base structure size makes no difference, since we are inserting, always close to its root and in exactly the same positions, the same 100,000 elements.

For the `LeftistHeap` implementation the observed behavior can be explained thus: the implementation abides by two invariants, `Heap` order and the Leftist property. The Leftist property states/guarantees that the rank of any left node (the underlying implementation is a binary tree) in the tree is greater than or equal to the rank of its right sibling (the rank of a node being the length of its right spine).

In following those invariants, the base data structure creation proceeds and results in exactly the same tree as for the `SkewHeap` implementation (Fig. A.19 (a)). That resulting tree's right nodes are all empty, with 0 rank, and consequently all inner nodes have rank 1.

When we start inserting the 100,000 elements, the invariants enforcement leads to, for the two initial elements (99999 and 99998), left/right branch exchanges and consequent node rank adjustments. But after that, the node ranks settle into a configuration, where the rank of a left node is always greater than the rank of its right sibling, that leads to the repeated insertion of the remaining elements into a single branch of the tree. Because we are inserting in decreasing order the algorithm also ends up inserting all remaining elements at the root of that branch. For a base of size n the resulting tree is depicted in Fig. A.20, where the blue path indicates the existing base structure, and the red path indicates the branch to which most of the 100,000 elements are inserted.

What we observed, leads to the conclusion that the resulting trees, for any base size n that we might consider, differ only by the height of a branch that is generated during the base data structure creation. The actual work incurred during our measurements is then irrespective of base data structure size.

For the `LazyPairingHeap` implementation the similar observed behavior can be explained as follows: the data structure underlying the `LazyPairingHeap` implementation is one that "fuses" the concepts of a list and binary tree. The base data structure creation proceeds in the manner already specified. For this implementation, for a base of size n, the resulting data structure is basically a list, from 0 to n.

The insertion of the 100,000 elements takes place by creating and merging (with possible swapping of tree branches) of list/tree structures/substructures, depending on the existing structure, on the ordering of values inserted relative to the values already present, and the invariant that must be maintained (that the left child of a tree portion of the structure may not be empty).

In the end, the resulting structure is a merge, of part of a "long" list generated during the base creation, and a more balanced tree created during the insertion of the 100,000 elements; once again because of our insertion methodology (decreasing order), from maximum to minimum, we end up inserting all elements at, or very close to, the top of the structure (start of the list/root of the tree). The fact that there is no need to travel deep into the data structure created as the base means that the work of insertion measured during the benchmark is independent of the base size.

Associative Collections

For `Associative Collections` two results, for operations `add` and `containsAll`, diverge from the general behavior.

For the `add` operation, the observed behavior is depicted in Fig. A.21.

For the `StandardMap` implementation, we can see that, the execution time mostly decreases as the base size increases. This behavior can be explained in the same way as for the results for the `StandardSet` implementation (see Page 46), because both these implementation rely on the same underlying data structure implementation, a binary search (without duplicates), size balanced, tree.

For the `AssocList` implementation, we can see that, the execution time only slightly increases as the base size increases. This is due to the `AssocList` underlying implementation. The `AssocList` implementation is based in a simple association list

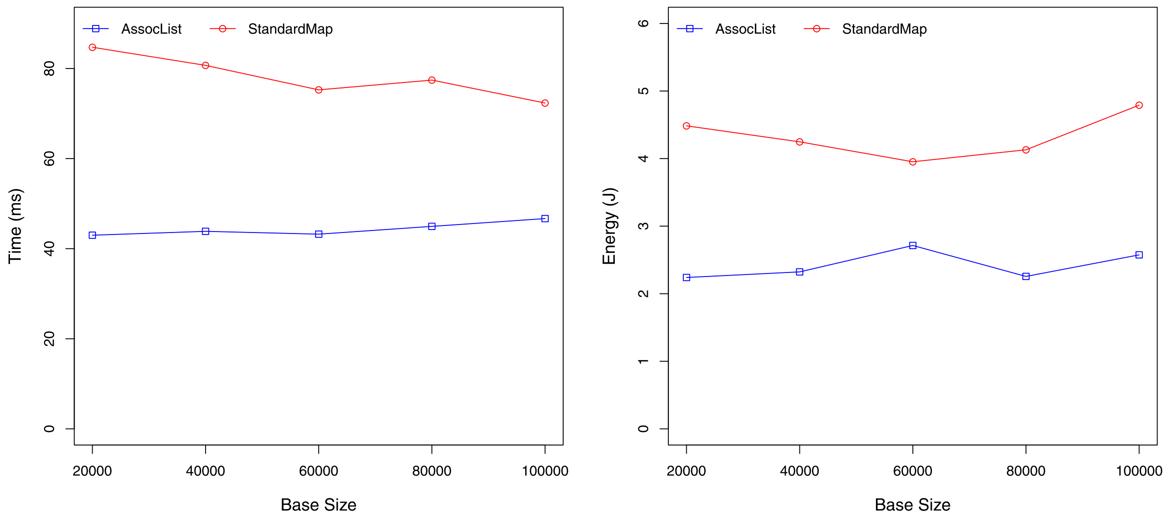


Fig. A21. Variability study for the add operation, for Associative Collections.

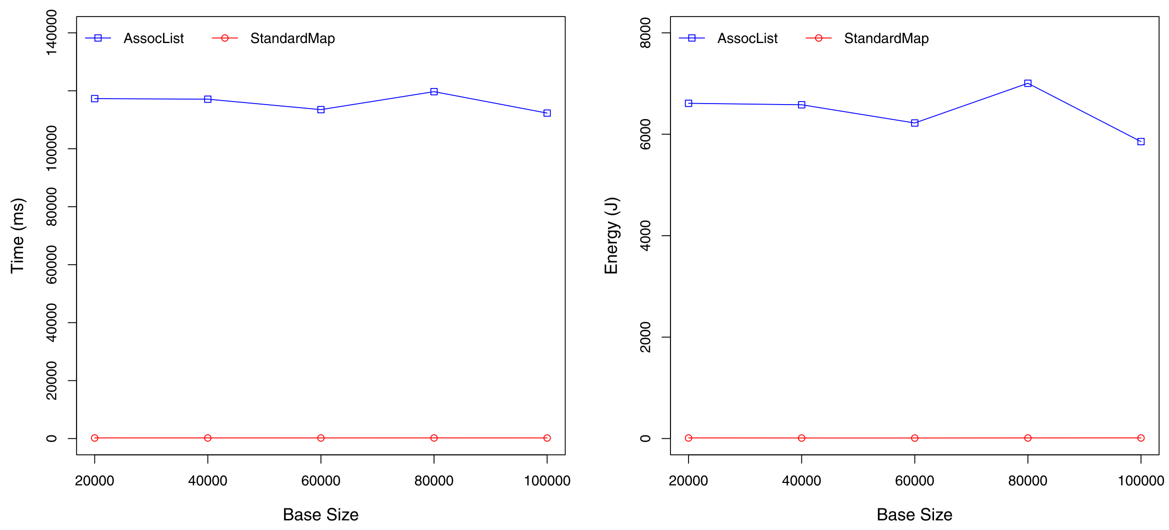


Fig. A22. Variability study for the containsAll operation, for Associative Collections.

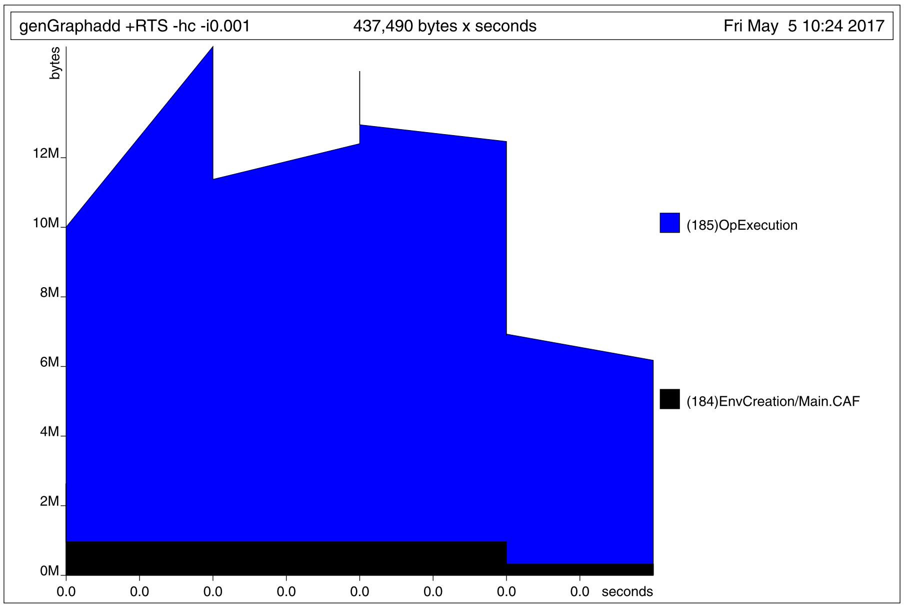


Fig. A23. Memory profiling for the execution of the add operation, for the Heaps abstraction, SplayHeap implementation, with increasing base structure size (step 1).

(a key, value, pair list) and the add operation is implemented via the insert function already provided in Edison.

This function is defined to be the “simple” (I) data constructor (from the Map type), that prepends the inserted key and element “pair” to a listlike structure without regard for duplicates. There should be, therefore, no big difference in execution time, for the insertion of n items, in a base data structure, irrespective of its size.

For the containsAll operation the observed behavior is depicted in Fig. A.22.

For the AssocList implementation, we can see that, the execution time is mostly constant, despite the increase in base size.

This may be explained by the fact that: the creation of a base data structure of size n results in a pure association list with key, and equal value, pairs from (0,0) to (n,n), in order. The data structure containing the (1000) elements to search for (ahead referred as the “searched for” map) is created in the same manner resulting in an Associative Collection with pairs (0,0) to (1000,1000), in that order.

The containsAll operation for Associative Collections, is based on the submap function provided by Edison; for the AssocList implementation this function operates on/iterates through the “searched for” map (which is always the same size, independent of the base data structure size), searching in the base structure for an element of that map; as can be concluded from the previously stated insertion strategy, this “map”, the list of pairs were searching for, is always found (in order) at the beginning of the base data structure.

We end up, then, searching through a limited “space” which is readily available. This implies that the workload is independent of the base data structure size. It is dependent on the size of the “searched for” map. Even so, the execution time is high for this implementation. This may be justified by the fact that during the operation the implementation tries to eliminate duplicates, repeatedly searching through the entire data structure which were searching for.

For the StandardMap implementation, we can see that: the execution time varies very little and is significantly lower than for

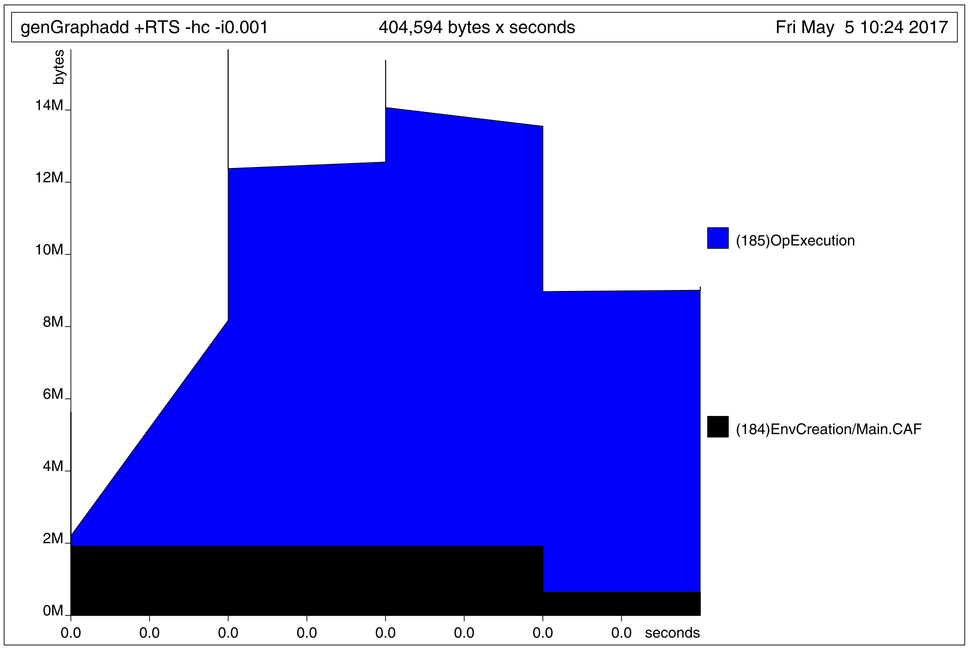


Fig. A24. Memory profiling for the execution of the add operation, for the Heaps abstraction, SplayHeap implementation, with increasing base structure size (step 2).

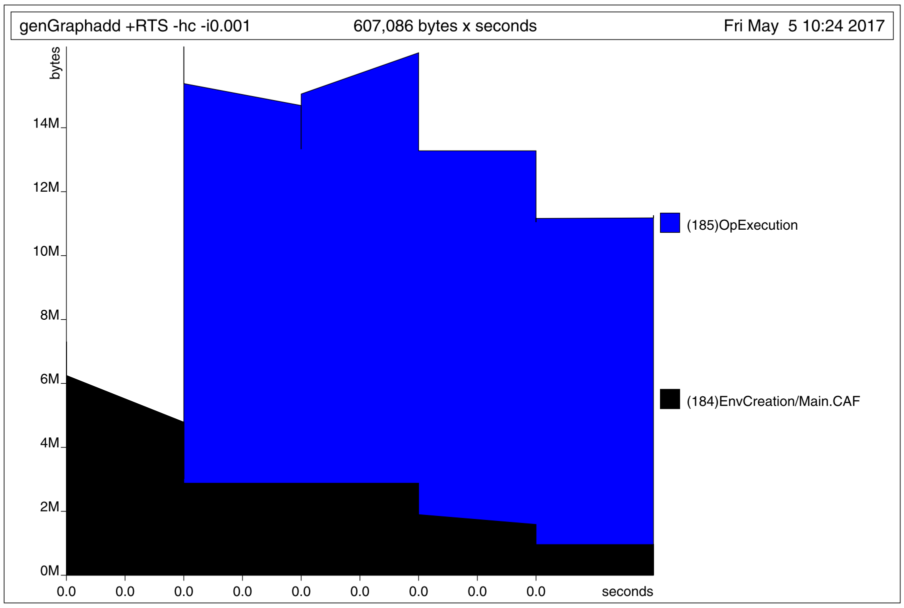


Fig. A25. Memory profiling for the execution of the add operation, for the Heaps abstraction, SplayHeap implementation, with increasing base structure size (step 3).

`AssocList`. This can be explained thus: the underlying implementation for `StandardMap` is the Data.Map standard `Haskell` module, which is itself implemented using a size balanced binary search tree; the `containsAll` operation is implemented through the submap function available in `Edison` which checks if a map m_1 is a submap of a map m_2.

In our case, we are searching for 1000 elements (our m_1) in the base data structures of sizes 20,000 to 100,000 (our m_2). The creation of m_1, and of each m_2 is carried out in the same manner as already detailed for other implementations/operations; it results in trees, balanced according to the Data.Map module's implementation: balancing takes into account two parameters, which as defined in the implementation results in trees whose left subtree generally contains more elements than its right subtree (e.g., the left subtree of m_1 has 744 elements).

In order to search for the elements of m_1 in m_2 the first thing the implementation does is to search, in m_2, for the element at the root of m_1, which is 744; because m_2 is a balanced binary search tree, this search happens in logarithmic time, and is not influenced by the base data structure sizes that we considered.

We know in advance that all the elements of m_1 will be found in m_2.

When navigating through m_2, when the node 744 is reached, the implementation will build two maps with the elements of m_2 which are smaller (lt), and bigger (gt), than 744; this means that lt will hold the exact same (744) elements regardless of the base structure size, and that its creation is also not affected by the data structure size (since all such elements are readily accessible from that node); recursively, the implementation will then determine if the left(right) submap of m_1, which contains all elements

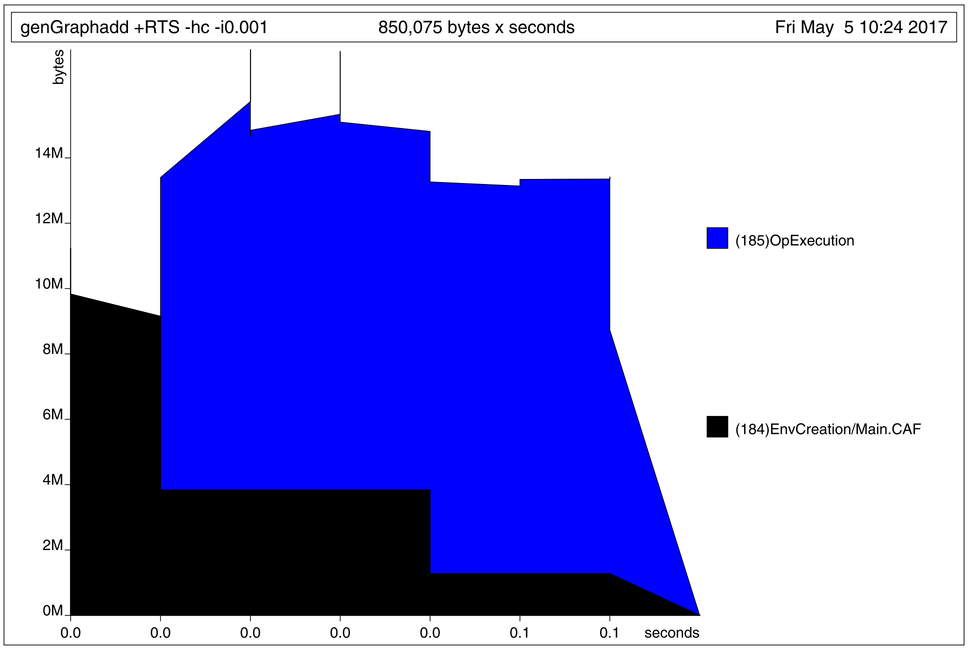


Fig. A26. Memory profiling for the execution of the add operation, for the Heaps abstraction, SplayHeap implementation, with increasing base structure size (step 4).

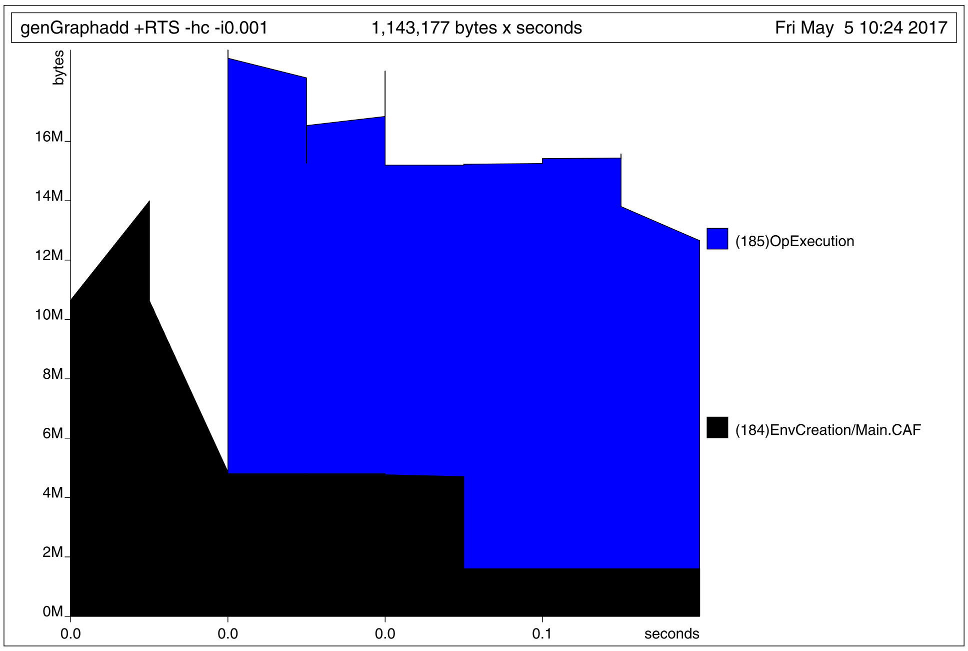


Fig. A27. Memory profiling for the execution of the add operation, for the Heaps abstraction, SplayHeap implementation, with increasing base structure size (step 5).

smaller(bigger) than the root, is a submap of lt(gt); in doing this, the right submap of m_1 is much smaller in size than gt (as m_1 is smaller than m_2); however, as explained before, the searching strategy will not be affected by this difference, even when the size of the base structures varies as in our experiment.

A2. Energy is not proportional to runtime

The vast majority of the consumption patterns that we have observed signal that the energy consumption proportionally follows the time consumption. This was actually also observed for the non-obvious cases, that we explained in the previous section, where the time consumption was not proportional to the increase in size of the base data structure.

The two single exceptions (out of 40 #operations*#abstractions) to this observation are described next.

Heaps

For the `add` operation, the results, that we have shown in Fig. A.18, for the `SplayHeap` implementation, we observe that: twice, the energy consumption decreases while the base size (and execution time) increases (from 20,000 to 40,000 and from 80,000 to 100000).

Associative Collections

For the add operation, for the StandardMap the observed behavior was already depicted in Fig. A.21, where we can see that: the energy consumption pattern, that leads to a concave upward curve, is dissimilar to the time consumption pattern.

Regarding these two cases, we admit that we were not able to provide a concrete justification for their occurrence. Our attempts to do so included trying to understand whether memory consumption, and its respective energy consumption (that we are not accounting for), could be affecting the observed (energy consumption) results. Our (admittedly vague) hypothesis here was that CPU-energy consumption could be decreasing/increasing as a consequence of energy consumption attributable to increasing/decreasing memory usage.

Since actually measuring memory-related energy consumption is outside the scope of this paper, we have used the Haskell profiler to estimate memory usage on these two cases, under the (initial) assumption that memory-related energy consumption is proportional to memory usage. As an example, we provide in Figs. A.23–A.27 the results we obtained for the add operation and the SplayHeap implementation, for the different base structure sizes that we considered.

While we strongly believe that the CPU/memory energy consumption relationship deserves further study, we admit that this initial effort provided no conclusive indication on either direction this relationship may point to.

References

Apfelmus, H., 2014. How lazy evaluation works in haskell. https://hackhands.com/lazy-evaluation-works-haskell/.
Apfelmus, H., 2015. Haskell's non-strict semantics - what exactly does lazy evaluation calculate? https://hackhands.com/non-strict-semantics-haskell/.
Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Mahaux, M., Penzenstadler, B., Rodríguez-Navas, G., Salinesi, C., Seyff, N., Venters, C.C., Calero, C., Koçak, S.A., Betz, S., 2015. Sustainability design and software: The karlskrona manifesto. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. IEEE Press, pp. 467–476.
Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., 2006. The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. ACM, pp. 169–190.
Brown, C., Loidl, H.-W., Hammond, K., 2012. ParaForming: forming parallel haskell programs using novel refactoring techniques. In: Proceedings of the 12th international conference on Trends in Functional Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 82–97.
Bruce, B.R., Petke, J., Harman, M., 2015. Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1327–1334.
Carção, T., 2014. Spectrum-based Energy Leak Localization. University of Minho, Portugal Master's thesis.
Cardelli, L., Wegner, P., 1985. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys (CSUR) 17 (4), 471–523.
Chandrakasan, A.P., Sheng, S., Brodersen, R.W., 1992. Low-power cmos digital design. IEEE Journal of Solid-State Circuits 27 (4), 473–484.
O. Corporation, 2018What's new in jdk 8? http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html.
deepseq package, 2018. http://hackage.haskell.org/package/deepseq.
David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C., 2010. Rapl: memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design. ACM, pp. 189–194.
David, T., Guerraoui, R., Trigonakis, V., 2013. Everything you always wanted to know about synchronization but were afraid to ask. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, pp. 33–48.
Desrochers, S., Paradis, C., Weaver, V.M., 2016. A validation of dram rapl power measurements. In: Proceedings of the Second International Symposium on Memory Systems. ACM, pp. 455–470.
Dig, D., Marrero, J., Ernst, M.D., 2011. How do programs become more concurrent: A story of program transformations. In: Proceedings of the 4th International Workshop on Multicore Software Engineering. ACM, pp. 43–50.
Dockins, R., 2018a. Edison, Haskell Communities and Activities Report 2009 https://www.haskell.org/communities/05-2009/html/report.html.
Dockins, R., 2018b. Edisonapi package. http://hackage.haskell.org/package/EdisonAPI-1.3.
Dockins, R., 2018c. Edisoncore package http://hackage.haskell.org/package/EdisonCore-1.3.
Hähnel, M., Döbel, B., Völp, M., Härtig, H., 2012. Measuring energy consumption for short code paths using rapl. SIGMETRICS Perform. Eval. Rev. 40 (3), 13–17.
Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M., 2005. Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, pp. 48–60.
Hejlsberg, A., Torgersen, M., 2018. Overview of c# 3.0. https://msdn.microsoft.com/en-us/library/bb308966.aspx.
Herlihy, M., Shavit, N., 2012. The Art of Multiprocessor Programming, 1st Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. Revised Reprint
Hindle, A., 2012. Green mining: A methodology of relating software change to power consumption. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, pp. 78–87.
JonesJr., D., Marlow, S., Singh, S., 2009. Parallel performance tuning for haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. ACM, pp. 81–92.
Kwon, Y.-W., Tilevich, E., 2013. Reducing the energy consumption of mobile applications behind the scenes. In: 2013 IEEE International Conference on Software Maintenance, pp. 170–179.
Lee, D.Y., 2011. A case study on refactoring in haskell programs. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 1164–1166.
Lewis, L., 2011. Java collection performance http://dzone.com/articles/java-collection-performance.
Li, H., Thompson, S., Reinke, C., 2005. The Haskell Refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science 141 (4), 29–34.
Lima, L.G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., Fernandes, J.P., 2016. Haskell in green land: analyzing the energy behavior of a purely functional language. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, pp. 517–528.
Lima, L.G.N.F., 2016. Understanding The Energy Behavior Of Concurrent Haskell Programs. Federal University of Pernambuco Master's thesis.
Liu, K., Pinto, G., Liu, Y.D., 2015. Data-oriented characterization of application-level energy optimization. In: Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, pp. 316–331.
Manotas, I., Pollock, L., Clause, J., 2014. Seeds: A software engineer's energy-optimization decision support framework. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 503–514.
Marlow, S., Brandy, L., Coens, J., Purdy, J., 2014. There is no fork: An abstraction for efficient, concurrent, and concise data access. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. ACM, pp. 325–337.
Melfe, G., Fonseca, A., Fernandes, J.P., 2018. Helping developers write energy efficient Haskell through a data-structure evaluation. In: Proceedings of the 6th International Workshop on Green and Sustainable Software. ACM, pp. 9–15.
Moura, I., Pinto, G., Ebert, F., Castor, F., 2015. Mining energy-aware commits. In: Proceedings of the 12th Working Conference on Mining Software Repositories. IEEE Press, pp. 56–67.
Murphy-Hill, E., Parnin, C., Black, A.P., 2009. How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 287–297.
Okasaki, C., 1999. Purely Functional Data Structures. Cambridge University Press.
Okasaki, C., 2001. An overview of edison. Electronic Notes in Theoretical Computer Science 41 (1), 60–73.
Oliveira, W., Oliveira, R., Castor, F., 2017. A study on the energy consumption of android app development approaches. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, pp. 42–52.
O'Sullivan, B., 2009. Criterion: Robust, reliable performance measurement and analysis. http://www.serpentine.com/criterion/.
Pankratius, V., Adl-Tabatabai, A.-R., 2011. A study of transactional memory vs. locks in practice. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. ACM, pp. 43–52.
Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J., 2017. Energy efficiency across programming languages: How do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. ACM, pp. 256–267.
Pierce, B., 2002. Type operators and kinding. Types and Programming Languages. MIT Press.
Pinto, G., Castor, F., Liu, Y.D., 2014. Understanding energy behaviors of thread management constructs. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. ACM, pp. 345–360.
Pinto, G., Liu, K., Castor, F., Liu, Y.D., 2016. A comprehensive study on the energy efficiency of java thread-safe collections. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 20–31.
Ribic, H., Liu, Y.D., 2016. Aequitas: Coordinated energy management across parallel applications. In: Proceedings of the 2016 International Conference on Supercomputing. ACM, pp. 4:1–4:12.
Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., Rajwan, D., 2012. Power-management architecture of the intel microarchitecture code-named sandy bridge. IEEE Micro 32 (2), 20–27.
Sahin, C., Pollock, L., Clause, J., 2014a. How do code refactorings affect energy usage? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, pp. 36:1–36:10.

Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., Clause, J., 2014b. How does code obfuscation impact energy usage? In: 30th IEEE International Conference on Software Maintenance and Evolution, pp. 131–140.
Sansom, P.M., Peyton Jones, S.L., 1995. Time and space profiling for non-strict, higher-order functional languages. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 355–366.
Scanniello, G., Erra, U., Caggianese, G., Gravino, C., 2015. On the effect of exploiting gpus for a more eco-sustainable lease of life. International Journal of Software Engineering and Knowledge Engineering 25 (01), 169–195.
Soares-Neto, F., 2014. Rewriting Concurrent Haskell Programs to STM. Federal University of Pernambuco Master's thesis.
Solernou, A., Thiyagalingam, J., Duta, M.C., Trefethen, A.E., 2013. The effect of topology-aware process and thread placement on performance and energy. In: 28th International Supercomputing Conference, pp. 357–371.
Tiwari, V., Malik, S., Wolfe, A., 1994. Power analysis of embedded software: A first step towards software power minimization. IEEE Trans. Very Large Scale Integr. Syst. 2 (4), 437–445.
Trefethen, A., Thiyagalingam, J., 2013. Energy-aware software: challenges, opportunities and strategies. Journal of Computational Science 4, 444–449.
Vásquez, M.L., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Penta, M.D., Poshyvanyk, D., 2014. Mining energy-greedy API usage patterns in android apps: an empirical study. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 2–11.
Weaver, V.M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., Moore, S., 2012. Measuring energy and power with papi. In: 2012 41st International Conference on Parallel Processing Workshops. IEEE, pp. 262–268.
Yuan, W., Nahrstedt, K., 2003. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. ACM, pp. 149–163.

Luís Gabriel Lima holds an MSc in Computer Science at the Center of Informatics of the Federal University of Pernambuco, Brazil, and currently works as a Software Engineer for Amazon Web Services. His main interests are functional programming, distributed systems, and building software at scale.

Francisco Soares-Neto is a full-time mentor of software development at the Apple Developer Academy - UFPE. His main research interests are programming languages and software engineering, specially functional programming. He holds a BEng degree in Computer Engineering from the University of Pernambuco, and an MSc degree in Computer Science from the Federal University of Pernambuco for research in refactoring concurrent Haskell programs.

Paulo Lieuthier holds a BSc in Computer Science at the Center of Informatics of the Federal University of Pernambuco, Brazil, and is currently employed as a Software Engineer. His main interests are functional, concurrent and performant programming and design of scalable systems.

Fernando Castor is an Associate Professor at the Informatics Center of the Federal University of Pernambuco, Brazil. His broad research goal is to help developers build more efficient software systems more efficiently. More specifically, he conducts research in the areas of Software Maintenance, Software Energy Efficiency, and Error Handling.

Gilberto Melfe Fascinated about computers since childhood. Holds a MSc in Informatics Engineering. His main interests are functional programming in Haskell and sustainability in ICT. Other areas he may like to investigate are OS, and hardware, development.

João Paulo Fernandes is an Assistant Professor at the Informatics Engineering Department of the University of Coimbra, Portugal, and a member of its Center for Informatics and Systems (CISUC). He pursues rigorous ways to reason about programming, particularly within the context of Green Computing, Functional Programming, Spreadsheets, Bidirectional Transformations and Software and Language Engineering.