Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language

Luís Gabriel Lima, Francisco Soares-Neto,
Paulo Lieuthier, Fernando Castor
Informatics Center
Federal University of Pernambuco (UFPE)
Recife, Brazil
{lgnfl, fmssn, pvjl, castor}@cin.ufpe.br

Gilberto Melfe*, João Paulo Fernandes*†
*LISP-Release, †HASLab/INESC TEC
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
gilbertomelfe@gmail.com, jpf@di.ubi.pt

***Abstract*—Recent work has studied the effect that factors such as code obfuscation, refactorings and data types have on energy efficiency. In this paper, we attempt to shed light on the energy behavior of programs written in a lazy purely functional language, Haskell. We have conducted two empirical studies to analyze the energy efficiency of Haskell programs from two different perspectives: strictness and concurrency. Our experimental space exploration comprises more than 2000 configurations and 20000 executions.**

We found out 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. In another benchmark, the latter primitive can yield up to 30% energy savings over the former. 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. 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. To support developers in better understanding this complex relationship, we have extended two existing performance analysis tools to also collect and present data about energy consumption.

I. Introduction

Energy-efficiency has concerned hardware and low-level software engineers for years [1], [2], [3]. However, the growing worldwide movement towards sustainability, including sustainability in software [4], 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 [5], Android API calls [6], object-oriented code refactorings [7], constructs for concurrent execution [8], and data types [9] 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 Vasquez et al. [6] 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 programs written in Haskell. 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 [10]. 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 [11]. Meanwhile, mainstream programming languages like Java and C# have adopted functional programming features such as lambdas [12], [13].

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. As for concurrency, previous work [8], [14] has demonstrated that concurrent programming constructs can influence energy consumption in unforeseen ways. In this paper, we attempt to shed more light on this complex subject. More specifically, we address the following high-level research question:

RQ. To what extent can we save energy by refactoring existing Haskell programs to use different data structure implementations or concurrent programming constructs?

To gain insight into the answer to this question, we conducted two complementary empirical studies. In the first one, we analyzed the performance and energy behavior of several benchmark operations over 15 different implementations of three different types of data structures. Even though Haskell has several implementations of well-known data structures [15], 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. To the best of our knowledge, this is the first study of its kind targeting Haskell's concurrent programming constructs. Overall, experimental space exploration comprises more than 2000 configurations and 20000 executions.

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. 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.

II. Related Work

Murphy-Hill et al. [16] provide an analysis on the use of refactoring. Their study indicates how refactoring is common, even if only executed manually. Dig and colleagues [17] 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. [18] present the Haskell Refactorer infrastructure to support the development of refactoring tools. Lee [19] used a case study to classify 12 types of Haskell refactorings found in real projects, mostly dealing with maintainability. Brown et al. [20] 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 [21] 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. [8] 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. [22] 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. 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 [23] reduced the energy consumption of mobile apps by offloading part of their computation transparently to programmers. Scanniello et al. [24] studied the migration of a performance-intensive system to an architecture based on GPU as a way to reduce energy waste. Moura et al. [25] 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. [26] 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.

III. Measuring Energy Consumption

This section presents the technology we used to measure the energy consumption of Haskell programs. In section III-A, we give a brief overview of our interface to gather energy information from Intel processors. In sections III-B and III-C 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) 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 IV-A), and a modified Criterion (Section III-B) library. Also, all experiments were performed with no other load on the OS.

A. RAPL

Running Average Power Limit (RAPL) [27] 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 [28]. Its precision and reliability has been extensively studied [29], [30].

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.

B. Criterion

Criterion [31] 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 as well as 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. That way, 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 or 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 [27]. 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.

C. GHC profiler

Currently, 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 [32]. 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.

IV. 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 IV-A, 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 IV-B.

A. A library of purely functional data structures

Our analysis relies on Edison, a fully mature and well documented library of purely functional data structures [15], [33]. Edison provides different functional data structures for

TABLE I
ABSTRACTIONS AND IMPLEMENTATIONS AVAILABLE IN EDISON.

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

implementing three types of abstractions: Sequences, Collections and Associative Collections. While these implementations are available in other programming languages, e.g., in ML [33], 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 [34].

In Table I, 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 [35] and EdisonAPI [36] 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.

Due to space limitations, we do not present here the complete lists of functions in the respective APIs. This information is available at green-haskell.github.io.

1) Collections*:* The Collections abstraction includes sets and heaps. While all implementations of these data structures share a significant amount of (reusable and uniform) methods[1], there are also methods 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.

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.

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 methods.

B. Benchmark

Following the approach considered in different studies [37], [38], [39], our benchmark is inspired by the microbenchmark to evaluate the run time performance of Java's JDK Collection API implementations [40]. The operations we consider are listed in Table II, and they all can be abstracted by the format:

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

[1]In this paper we refer to functions and methods interchangeably

TABLE II
BENCHMARK OPERATIONS.

iters	*operation*	*base*	*elems*
1	add	100000	100000
1000	addAll	100000	1000
1	clear	100000	n.a.
1000	contains	100000	1
5000	containsAll	100000	1000
1	iterator	100000	n.a.
10000	remove	100000	1
10	removeAll	100000	1000
10	retainAll	100000	1000
5000	toArray	100000	n.a.

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, 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).

C. 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 behaviour. 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 III 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 =
  let elem =  m + n - 1
      cons = if even n then rcons else lcons
  in add (elem `cons` seq) (n-1) m
```

TABLE III
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 [41] in many definitions. 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) (removeNTimes h ( n - 1 ))
```

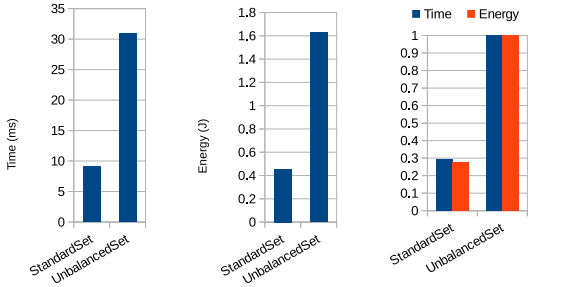
We have tried to follow as much as possible the data structure sizes and number of iterations suggested by the benchmark described in the previous section. In a few cases, however, we needed to simplify concrete operations for specific abstractions. This simplification was performed whenever a concrete 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 reducing the number of iterations in half. 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 IV lists the operations whose inputs or number of iterations were simplified. The underlined elements of this table are the ones that differ from the original benchmark.

D. Results

We analyse the results we obtained following the methodology described in the previous section. Due to space limitations, we are not able to include the observed results for all operations on all abstractions. They are available at the companion website, at green-haskell.github.io.

TABLE IV
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



(a) (b) (c)

Fig. 1. Results of clear operation for Sets

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 Figure 1. In Figures 1 (a) and (b) we compare the absolute values obtained for the runtime execution and energy consumption, respectively. In Figure 1 (c) we compare the proportions of time and energy comsumption: 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 Figures 2 (a) and 2 (b), with the comparison between proportions of runtime and energy consumption for add and toList, 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 Figure 3 (a)), to 99.999% for retainAll. For the

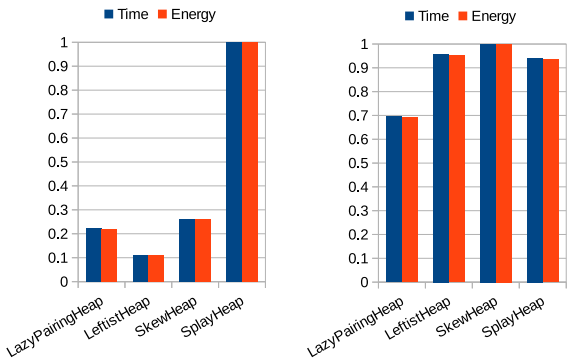


Fig. 2. Results of add and toList operations for Heaps

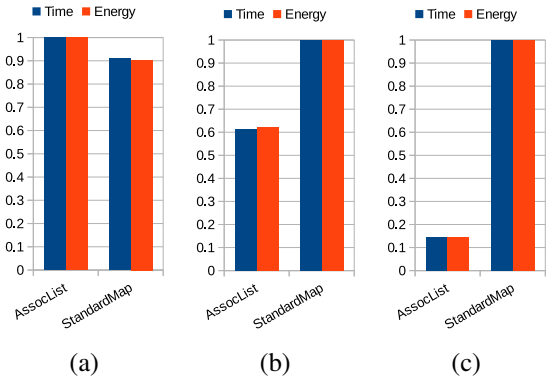


Fig. 3. Results of addAll, add and iterator for Associative Collections

add and iterator operations, illustrated in Figures 3 (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 Figure 4 for the remove operation. The observed proportions across all operations and implementations differ at most in 1.9%, for the add operation.

V. 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 motivated by 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 V-A). Section V-B then describes the set of benchmarks that we used. Since we analyzed multiple variants of each benchmark,

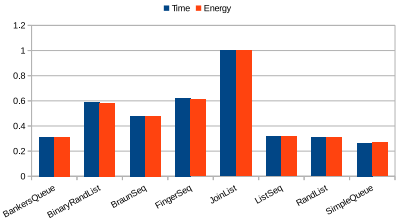


Fig. 4. Results of remove operation for Sequences

Section V-C explains how we adapted them to produce these variants. Finally, Section V-D presents the results of the study.

A. 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.

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 [44], 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

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

TABLE V
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[2]. 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 [42].
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 [43].

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.

B. Benchmark

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 Benchmarks 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. Table V 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.

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

[4]http://rosettacode.org/

C. 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: $\{1, 2, 4, 8, 16, 20, 32, 40, 64\}$. Where 20 and 40 are the number of physical and virtual cores, respectively.

D. Results

In this section, we report the results of our experiments with concurrent Haskell programs. The results are presented in Figure 5. Here, the odd rows are energy consumption results, while the even rows are the corresponding running time results. We omitted the experiments using 64 capabilities in order to make the charts more readable. The charts including this

configuration as well as all the data and source code used in this study are available at green-haskell.github.io.

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

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

Faster is not always greener. Overall, the shapes of the curves in Figure 5 are similar. Although, for 6 of our 9 benchmarks, in at least two variants of each one, there are moments where faster running 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.

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 6 times more energy. However, for the chameneos-redux benchmark, the forkIO-TMVar variant consumes 2.4 times more energy than the best variant, forkIO-MVar. This example is particularly interesting because these two benchmarks have similar characteristics. Both dining-philosophers and chameneos-redux are synchronization-intensive benchmarks and both have a fixed number of worker threads. Even in a scenario like this, using the same constructs can lead to discrepant results.

Choosing more capabilities than available CPUs is harmful. The performance of most benchmarks is severely impaired by using more capabilities than the number of available CPUs. In chameneos-redux, for example, moving from 40 to 64 capabilities can cause a 13x slowdown. This suggests that the Haskell runtime system was not designed to handle cases where capabilities outnumber CPU cores. In fact, this assumption makes sense as the official GHC documentation recommends the number of capabilities to match the number of CPU cores. However, the documentation leaves as an open question if virtual cores should be counted. In our experiments, the performance almost never improves after 20 capabilities. So developers should be careful when using more capabilities than available physical CPU cores as it can degrade performance.

VI. Discussion

Generally, especially in the sequential benchmarks, high performance is a proxy for low energy consumption. Our first study (Section IV) 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 construct is very simple in Haskell. Functions forkOn, forkIO, and forkOS take a computation of type IO as parameter and produce results of the same type. Thus, the only difficulty is in determining on which capability a thread created via forkOn will run. This is good news for developers and maintainers. Considering the 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 [45]. 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.

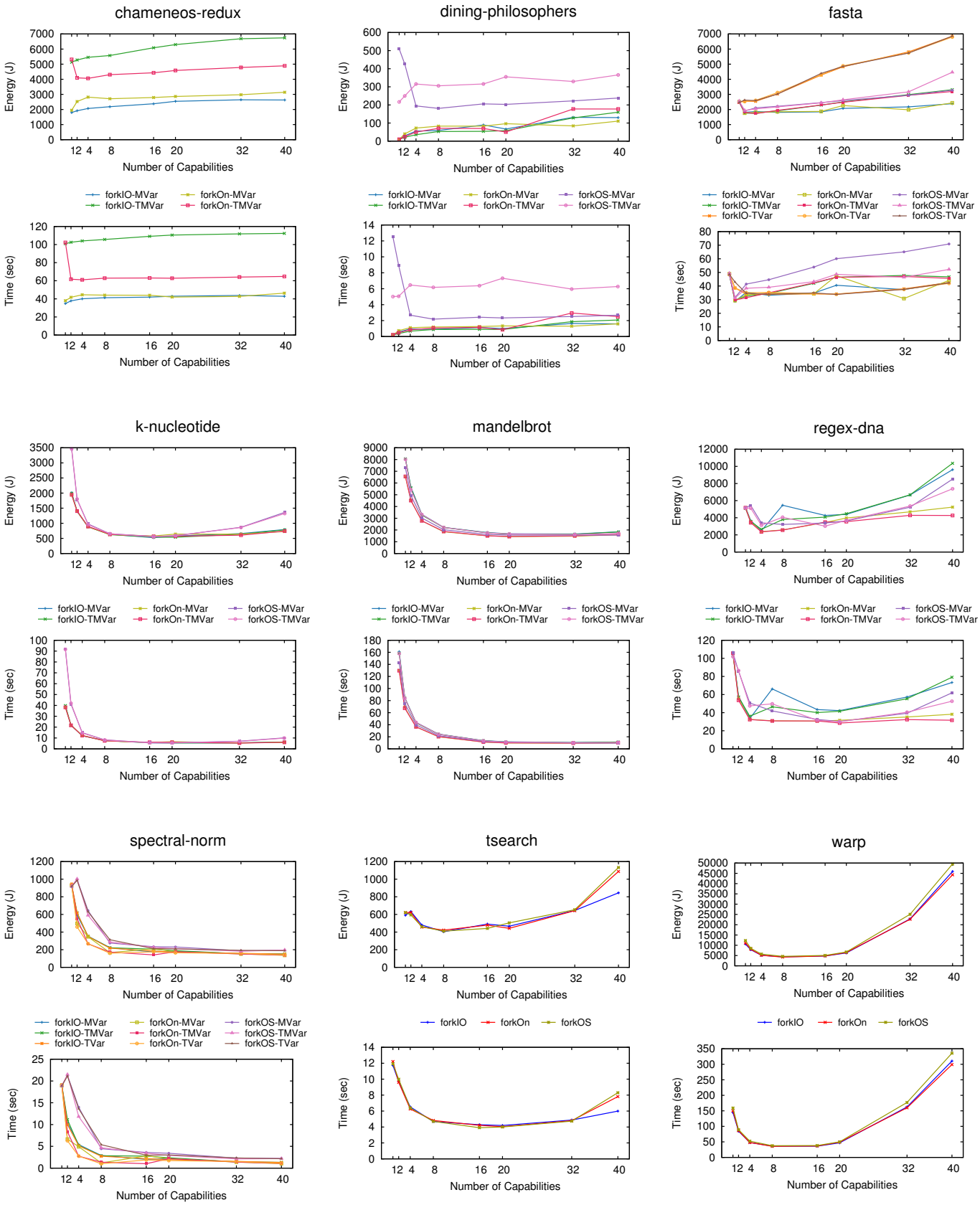


Fig. 5. Energy/Time with alternative concurrency abstractions and varying the number of capabilities

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

Although there is no overall winner, there is a more or less clear loser, when thread management construct have a strong impact on energy: forkOS. In only one of the benchmarks the forkOS variants did not have the worst energy consumption and worst performance: regex-dna. According to the Haskell documentation [46], *"Using forkOS instead of forkIO makes no difference at all to the scheduling behavior of the Haskell runtime system"*. If that is the case, the extra time and energy consumption stem from the need to switch OS threads to execute work passed to a call to forkOS. This overhead does not exist for forkOn and forkIO.

VII. 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 [3] and processor and interconnect layouts [47] 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 runing Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25) and GHC 7.10.2. Figure 6 shows the results of mandelbrot running on this i7 machine. The results show analogous trends in which the curves have similar shapes to the results of Figure 5. The same trend can be observed for the remaining benchmarks.

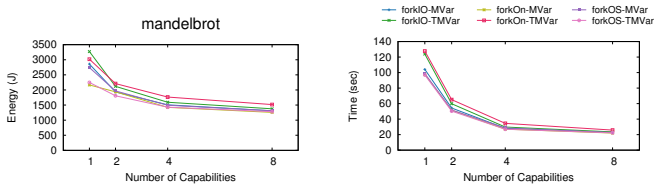


Fig. 6. Energy/Time on Alternative Platform

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

One further threat is related to our measurement approach We have employed RAPL to measure energy consumption. Thus, the results could be different for external measurement equipment. Nonetheless, previous work [48] has compared the accuracy of RAPL with that of an external energy monitor and the results are consistent.

VIII. 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.

IX. Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. Fernando is supported by CNPq/Brazil (304755/2014-1, 487549/2012-0 and 477139/2013-2), FACEPE/Brazil (APQ- 0839-1.03/14) and INES (CNPq 573964/2008-4, FACEPE APQ-1037-1.03/08, and FACEPE APQ-0388-1.03/14). Any opinions expressed here are from the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 4, pp. 473–484, Apr 1992.

[2] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, Dec 1994.

[3] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. ACM, 2003, pp. 149–163.

[4] C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, M. Mahaux, B. Penzenstadler, G. Rodríguez-Navas, C. Salinesi, N. Seyff, C. C. Venters, C. Calero, S. A. Koçak, and S. Betz, "The karlskrona manifesto for sustainability design," *CoRR*, vol. abs/1410.6968, 2014.

[5] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How does code obfuscation impact energy usage?" in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 131–140.

[6] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in android apps: an empirical study," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, 2014, pp. 2–11.

[7] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 36:1–36:10.

[8] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 345–360.

[9] K. Liu, G. Pinto, and Y. Liu, "Data-oriented characterization of application-level energy optimization," in *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, vol. 9033, 2015, pp. 316–331.

[10] B. Pierce, "Type operators and kinding," in *Types and Programming Languages*. MIT Press, 2002, ch. 29.

[11] S. Marlow, L. Brandy, J. Coens, and J. Purdy, "There is no fork: An abstraction for efficient, concurrent, and concise data access," *SIGPLAN Not.*, vol. 49, no. 9, pp. 325–337, Aug. 2014.

[12] O. Corporation. What's new in jdk 8? [Online]. Available: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

[13] A. Hejlsberg and M. Torgersen. Overview of c# 3.0. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb308966.aspx

[14] A. E. Trefethen and J. Thiyagalingam, "Energy-aware software: Challenges, opportunities and strategies," *Journal of Computational Science*, vol. 4, no. 6, pp. 444 – 449, 2013.

[15] C. Okasaki, "An overview of edison," *Electronic Notes in Theoretical Computer Science*, vol. 41, no. 1, pp. 60–73, 2001.

[16] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5–18, Jan 2012.

[17] D. Dig, J. Marrero, and M. D. Ernst, "How do programs become more concurrent: a story of program transformations," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE. ACM, 2011.

[18] H. Li, S. Thompson, and C. Reinke, "The Haskell Refactorer, HaRe, and its API," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 4, pp. 29–34, dec 2005.

[19] D. Y. Lee, "A case study on refactoring in haskell programs," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE. ACM, 2011.

[20] C. Brown, H.-W. Loidl, and K. Hammond, "ParaForming: forming parallel haskell programs using novel refactoring techniques," in *Proceedings of the 12th international conference on Trends in Functional Programming*, ser. TFP'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 82–97.

[21] A. Hindle, "Green mining: A methodology of relating software change to power consumption," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012, pp. 78–87.

[22] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: ACM, 2014, pp. 36:1–36:10.

[23] Y.-W. Kwon and E. Tilevich, "Reducing the energy consumption of mobile applications behind the scenes," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 170–179.

[24] G. Scanniello, U. Erra, G. Caggianese, and C. Gravino, "On the effect of exploiting gpus for a more eco-sustainable lease of life," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 01, pp. 169–195, 2015.

[25] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining energy-aware commits," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 56–67.

[26] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15. New York, NY, USA: ACM, 2015, pp. 1327–1334.

[27] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*. IEEE, 2010, pp. 189–194.

[28] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 262–268.

[29] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, no. 2, pp. 20–27, 2012.

[30] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[31] B. O'Sullivan. (2009) criterion: Robust, reliable performance measurement and analysis. [Online]. Available: http://www.serpentine.com/criterion/

[32] P. M. Sansom and S. L. Peyton Jones, "Time and space profiling for non-strict, higher-order functional languages," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 355–366.

[33] C. Okasaki, *Purely functional data structures*. Cambridge University Press, 1999.

[34] R. Dockins. Edison, Haskell Communities and Activities Report 2009. [Online]. Available: https://www.haskell.org/communities/05-2009/html/report.html

[35] ——. Edisoncore package. [Online]. Available: http://hackage.haskell.org/package/EdisonCore-1.3

[36] ——. Edisonapi package. [Online]. Available: http://hackage.haskell.org/package/EdisonAPI-1.3

[37] I. Manotas, L. Pollock, and J. Clause, "Seeds: A software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 503–514.

[38] T. Carção, "Spectrum-based energy leak localization," Master's thesis, University of Minho, Portugal, 2014.

[39] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java thread-safe collections," *Journal of Systems and Software*, 2016, to appear.

[40] L. Lewis. (2011) Java collection performance. [Online]. Available: http://dzone.com/articles/java-collection-performance

[41] deepseq package. [Online]. Available: http://hackage.haskell.org/package/deepseq

[42] V. Pankratius and A.-R. Adl-Tabatabai, "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, 2011, pp. 43–52.

[43] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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, 2006, pp. 169–190.

[44] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the 10th PPoPP*, 2005.
[45] F. Soares-Neto, "Rewriting concurrent haskell programs to stm," Master's thesis, Federal University of Pernambuco, February 2014.
[46] forkos documentation. [Online]. Available: https://hackage.haskell.org/package/base-4.8.1.0/docs/Control-Concurrent.html#v:forkOS
[47] A. Solernou, J. Thiyagalingam, M. C. Duta, and A. E. Trefethen, "The effect of topology-aware process and thread placement on performance and energy," in *28th International Supercomputing Conference*, 2013, pp. 357–371.
[48] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012.