The First Workshop on Software Engineering for Parallel Systems (SEPS 2014)
Portland, Oregon, United States
October 21, 2014

Characterizing the Energy Efficiency of Java's Thread-Safe Collections in a Multi-Core Environment

Gustavo Pinto[a,*], Fernando Castor[a]

[a]*Federal University of Pernambuco, Recife, PE, 50.740-560, Brazil*

Abstract

Java programmers are served with numerous choices of collections, varying from simple sequential ordered lists to more sophisticated, thread-safe, and highly scalable hashtable implementations. These choices are well-known to have different characteristics in terms of performance, scalability, and thread-safety, and most of them are well studied. This paper analyzes an additional dimension, *energy efficiency*. Through an empirical investigation of 16 collection implementations grouped under 3 commonly used collections (`Lists`, `Sets` and `Maps`), we show that small design decisions can greatly impact energy consumption. The study serves as a first step toward understanding the energy efficiency of Java collections on parallel architectures.

Keywords: Energy Efficiency, Performance, Java Collections

1. Introduction

A question that often rises in software development forums is: "since Java has so many collection implementations, which one is more suitable to my problem?"[1]. Answers to this question come in different flavors: these collections serve for different purposes and have different characteristics in terms of performance, scalability and thread-safety. Developers should consider these characteristics in order to make judicious design decisions about which implementation best fits their problems. In this study, we consider one additional attribute: *energy efficiency*.

Traditionally addressed by hardware-level (*e.g.*, [1, 2]) and system-level approaches (*e.g.*, [3, 4]), energy optimization is gaining momentum in recent years by focusing on application development (*e.g.*, [5, 6]). This crescent interest is in part due to recent studies that have provided empirical evidences that even great strides can be achieved when software engineers start to play the role of reducing energy consumption through their high-level design and implementation decisions [7, 8]. However, in order to find energy-efficient solutions, developers sometimes make use of conventional wisdom, consult software development forums and blogs, or simply search online for "tips and tricks". Unfortunately, many of the available suggestions are not supported by empirical evidence. Also, as pointed out by recent research, some of these guidelines are often anecdotal or even incorrect [9].

Moreover, there is considerable evidence that many users complain about battery usage when writing reviews about their apps [10]. Battery consumption can play an important role on the decision to adopt an app. However, even though there are existing tools that can help developers to gain insight into the energy usage of their applications [11, 12], these tools do not provide direct guidance on how to improve the overall energy con-

*Corresponding author
Email addresses: `ghlp@cin.ufpe.br` (Gustavo Pinto), `castor@cin.ufpe.br` (Fernando Castor)

[1] http://stackoverflow.com/search?q=which+data+structure+use+java+is:question

sumption of an application; that is, they do not address the gap between understanding where energy is consumed and understanding how the code can be changed in order to reduce energy consumption. This fact provide incentives to researchers to conduct new empirical studies on the subject.

Unfortunately, despite its importance, there is a gap in the literature of empirical studies tackling the problem of understanding the energy consumption impact of using different Java collections running on parallel architectures [13]. We believe this is an important topic that deserves more investigation due to at least three reasons: (1) data structures are one of the most important building blocks of computer programming; (2) not only high-end servers but also desktop machines, smartphones and tablets need concurrent programs to make the best use of their multi-core hardware; and (3) a CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2) [14].

This paper takes a step towards remedying this problem. We present an empirical study consisting on the evaluation of performance and energy consumption characteristics of 16 Java collection implementations grouped by 3 well-known collections: `List`, `Set`, and `Map`. The goal of this work is to obtain a deeper understanding of the energy consumption behavior of the Java concurrent collections. Trough an empirical exploration conducted in a multi-core environment, we correlate energy behaviors of different thread-safe implementations of Java collections and their knobs. We demonstrate that several factors can impact energy efficiency and performance in different ways. The main findings of this study are the following:

- Different implementations of the same collection exhibit very different energy consumption behavior. For example, a removal operation on a `Collections.synchronizedSet()` can be more than 4 times more expensive than a traversal on a `ConcurrentHashSetV8`.

- Different operations on the same implementation also behave differently. For example, removal operations in a `ConcurrentSkipListMap` can be more than 4 times expensive than an insertion. Also, for `CopyOnWriteArraySet`, an insertion consumed three order of magnitude more than a read. These results suggest that, to select an appropriate collection implementation, developers must carefully consider how it will be used.

- Execution time can safely be used as a proxy for energy consumption when dealing with `Lists` and `Sets`. The same it not always true for `Maps`.

- Faster is not a synonym for greener. We have observed cases where a high-performance implementation consumes more energy than a single-lock based one.

In this study, we examine 3 basic operations, analyze energy-performance trade-offs and stick the to comparing implementations of the same collections. We believe that cross-collection comparisions would not be very interesting, since they serve for different purposes. With the results of this study, we believe we can influence the high-level programming decisions of next generation of energy-aware programmers.

2. Related Work

The energy impacts of different design decisions made by software engineers have been previously investigated in several empirical studies. These studies analyzed a number of factors, varying from sorting algorithms [15], constructs for managing concurrent execution [6], design patterns [16], refactoring [8], cloud offloading [17, 7, 18], VM services [19], code obfuscation [20], among many others. Zhang *et al.* [18] presented a mechanism for automatically refactoring an Android app into one implementing the on-demand computation offloading design pattern, which can transfer some computation-intensive tasks from a smartphone to a server so that the task execution time and battery power consumption of the app can be reduced significantly. Cao *et al.* [19] described how different VM services (such as the Just-In-Time compiler, interpretation and/or the garbage collector) consumes in energy consumption. They observed that together these services impose substantial energy and performance costs, ranging from 10% to over 80%. In contrast, Li *et al.*[5] presented an evaluation of a set of programming practices suggested in the official Android developers web site. They observed that although some practices, such as the network packet size, can provide interesting degrees of savings, while some others, such as limiting memory usage, had a very minimal impact on energy

usage. Finally, the work of Vallina-Rodriguez *et al.* [21] presents a survey on general solutions for energy efficiency on mobile devices at the software level. These solutions vary from operating system solutions to energy savings via process migration to the cloud and protocol optimizations.

The performance of data structures is also an active area of research, with great improvements in lock-free data structures [22], spatial data structures [23], dynamic-sized data structures [24], among many others. The Java collections are also focus of several studies [25, 26, 27]. Our work and related work cited here are complementary. Together, they attempt to understand the performance of different data structures. However, all of the works mentioned above related to data structures do not provide general high level guidance to developers in terms of energy efficiency practices in programming.

To the best of our knowledge, only two studies dealt with the topic of understanding how energy consumption changes when developers employ different collections [28, 13]. In the first study, Manotas *et al.* [28] focus on a framework used to optimize energy consumption by automatically selecting the most energy-efficient collection implementation. This framework alternates the implementations and measures the energy consumption at runtime. In this study, however, the authors do not analyze their subjects in a multi-core environment, and also they do not discuss the impact of different operations (such as reads, insertions and removals) on energy consumption. In the study of Hunt *et al.* [13], the authors provided a comprehensive overview in terms of energy, power and performance of three data structures (a simple FIFO, a double-ended queue, and a sorted linked list). The authors also demonstrated a strong correlation between the performance of a data structure and its total energy consumption. However, we believe that our work greatly extends their work, considering 3 groups of collections, implemented by 16 classes. We also analyze the cost of read, insertion and removal operations, in addition to `Map` implementations, which are not covered by the study of Hunt *et al.* [13].

3. Study Setup

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

3.1. Research Questions

Our research is motivated by the following research questions:

RQ1. Do different implementations of the same collection have different impacts on energy consumption?

RQ2. Do different operations in the same implementation of a collection consume energy differently?

The goal of this study is to answer these research questions. To achieve this goal, we performed an experimental space exploration over well-known thread-safe Java collections.

3.2. Benchmarks

The benchmarks used in this study consist of 16 commonly used collections available in the Java programming language. Our focus is on the thread-safe implementations of the data structures. Hence, for each data structure, we selected a single non-thread-safe implementation to serve as a baseline. For each one of them, we analyzed insertion, removal and traversal operations. We grouped these implementations by their collections.

`Lists (java.util.List)`: `Lists` are ordered collections that allow duplicate elements. Using this collection, programmers can have precise control over where an element is inserted in the list. The programmer can access all elements using their indexes, or traverse the elements using an `Iterator`. Several implementations of this collection are available in the Java language. We used `ArrayList`, which is not thread-safe, as our baseline. We also used the following thread-safe `List` implementations: `Vector`, `Collections.synchronizedList()`, and `CopyOnWriteArrayList`. The latter was introduced in Java 5 Concurrency API. It achieves thread-safety in a slightly different way than `Vector`. This class by creates a copy of the underlying `ArrayList` whenever a mutation operation (*e.g.*, using the `add()` or `set()` methods) is invoked.

`Sets (java.util.Set)`: As its name suggests, the `Set` collection models the mathematical set abstraction. Unlike `Lists`, `Sets` do not count

duplicate elements, and are not ordered. Thus, the elements of a set cannot be accessed by their indexes, and traversals are only possible using an `Iterator`. Among the available implementations, we used `LinkedHashSet`, which is not thread-safe, as our baseline. We also used the following thread-safe `Set` implementations: `CopyOnWriteArraySet`, `Collections.synchronizedSet()`, `ConcurrentSkipListSet`, `ConcurrentHashSet`, and `ConcurrentHashSetV8`. Although there is no `ConcurrentHashSet` implementation available in the JDK, we can mimic its behavior by using a `Collections.newSetFromMap(new ConcurrentHashMap<Object,Boolean>())`. The resulting `Set` displays the same ordering, scalability in the presence of multiple thread, and performance characteristics as the backing map.

`Maps (java.util.Map)`: `Maps` are objects that map keys to values. The keys of a map cannot be duplicated, and are associated with at most one value. The values can be duplicated. From the available maps, we used `LinkedHashMap`, which is not thread-safe, as our baseline. We did not used `HashMap` as our baseline, because it entered into an infinite loop when performing our experiments[2]. We also used the following thread-safe `Map` implementations: `Hashtable`, `Collections.synchronizedMap()`, `ConcurrentSkipListMap`, `ConcurrentHashMap`, and `ConcurrentHashMapV8`. The difference between the two `ConcurrentHashMaps` is that the latter is an optimized version released in Java 1.8, while the former is the version present in the JDK until Java 1.7. All these `Map` implementations offer mostly the same functionalities. The most important difference, however, is the order in which iteration through the entries will happen. For instance, while `LinkedHashMap` iterates in the order in which the elements are added into the map, a `Hashtable` makes no guarantees about the iteration order.

3.3. Experimental Environment

All experiments were conducted on a machine with 2×16-core AMD Opteron 6378 processors (Piledriver microarchitecture) running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64), 64GB of DDR3 1600 memory, Oracle HotSpot 64-Bit server VM, and JDK version 1.7.0_11, build 21. When we performed the experiments with `Sets` and `Maps`, we employed the jsr166e library[3], which contains the `ConcurrentHashMapV8` implementation. Thus, these experiments do not need to be executed under Java 1.8.

All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We chose the last three runs because, according to a recent study, JIT execution tends to stabilize in the latter runs [6].

Energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere). Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples multiplied by $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other application) running. The reported results are the measured results *modulo* the "base" energy consumption.

4. Study Results

In this section, we report the results of our experiments. In **RQ1** and **RQ2**, we fixed the number of threads in 32 and, for each group of collections, we performed and measured insertion and traversal operations. Each thread inserts 100,000 elements. To avoid duplicate elements, we used the resulting string *thread-id* + "-" + *current-index* as the element to be added. The removal operation occurs in place; that is, there is no need to traversal the data structure.

[2] A possible explanation can be found here: http://mailinator.blogspot.com/2009/06/beautiful-race-condition.html

[3] Source code available at: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166e/

Figure 1 shows the overall view of our experimental results. The figure shows the energy consumption (bars) and execution time (line). Each bar represents one collection. The figures in the top, middle and bottom represent the collection implementations of the `List`, `Set` and `Map` collections, respectively. Figures in the left show traversal operations whereas figures in the right show insertion operations. We did not show the figures for `CopyOnWriteArrayList` and `CopyOnWriteArraySet` because they are outliers and biased the meaning of the figures.

We now describe the results in terms of each group of collection.

Lists. Taking into consideration the implementations of the `List` collection, we can see that, for insertion operations, `ArrayList` is the most energy efficient. When comparing the thread-safe implementations, `Vector` consumes 1.30x less energy than `Collections.synchronizedList()` (1.24x for execution time). On the other hand, `CopyOnWriteArrayList` consumes about 152x more energy than `Vector`. This is because, for each new element added to the list, `CopyOnWriteArrayList` needs to synchronize and create a fresh copy of the underlying array using the `System.arraycopy()` method. As discussed elsewhere [6], even though the `System.arraycopy()` behavior can be noticeable in sequential applications, it is more evident in highly parallel applications, when several processors are busy making copies of the data structure, preventing them from doing important work. Although this behavior makes this implementation thread-safe, it is ordinarily too costly to maintain the collection in a highly concurrent environment where insertions are not very rare events.

The traversal operations also incur some trade-offs. The traversal operations described here are performed using a top-level loop over the collection, accessing each element by its index using the `List.get(Object o)` method[4]. In this configuration, the `Vector` implementation presents the worst result among the benchmarks: it consumes 14.58x more energy and 7.9x more time than the baseline. One of the reasons for that is because the `Vector` and `Collection.synchronizedList()` implementations need to synchronize in traversal operations. In contrast, the `CopyOnWriteArrayList` implementation is more efficient than `Vector` for traversal operations, consuming 46.38x less energy than `Vector`. We also observed that, when the upper bound limit need to be computed in each iteration, for instance, using `for (int i=0; i<list.size(); i++)`, the `Vector` implementation consumed about twice as much as it consumed when using this limit is computed only once (1.98x more energy and 1.96x more time), for instance, using `int size = list.size(); for(int i=0; i<size; i++)`.

Moreover, in order to understand the `get()` behavior presented in the `Collections.synchronizedList()` implementation, it is important to take into consideration some internal implementation details. The `Collections.synchronizedList()` method creates an instance of the `SynchronizedList` class, which is a synchronized proxy for any `List` implementation. This class can be seen as a relative of the `Vector` one, except for the fact that the latter synchronizes in the `Iterator`, whereas the former does not. As observed in Figure 1, `Collections.synchronizedList()` performs much better than a `Vector`. We believe that the main reason for this redundancy is backward compatibility with Java code developed for old versions of Java. Before Java 1.2, the `Collections` class was not part of standard JDK/JRE environment.

Nowadays, the main difference between `Vector` and `SynchronizedList` is the way of use. By using `Collections.synchronizedList()`, the programmer creates a wrapper around the current `List` implementation, which does not need to copy data to another data structure. It is appropriate in cases where the programmer wants to use a `LinkedList` as opposed to an `ArrayList`. Using a `Vector`, on the other hand, it is not possible to keep an alternative underlying structure (such as `LinkedList`). We cannot perform removals using `Iterator`s because they are "fast-fail", that is, they fail as soon as they realize that the underlying structure has been modified since iteration begun. Such changes mean adding, removing, or updating any element from a collection while one thread is iterating over that collection. When it happens, the `Iterator` throws a `ConcurrentModificationException`.

We also analyzed traversal operations when the programmer iterates using an *enhanced for*

[4]We can not reproduce this experiment using the `Set` implementations, because this collection does not provide the `get()` method.

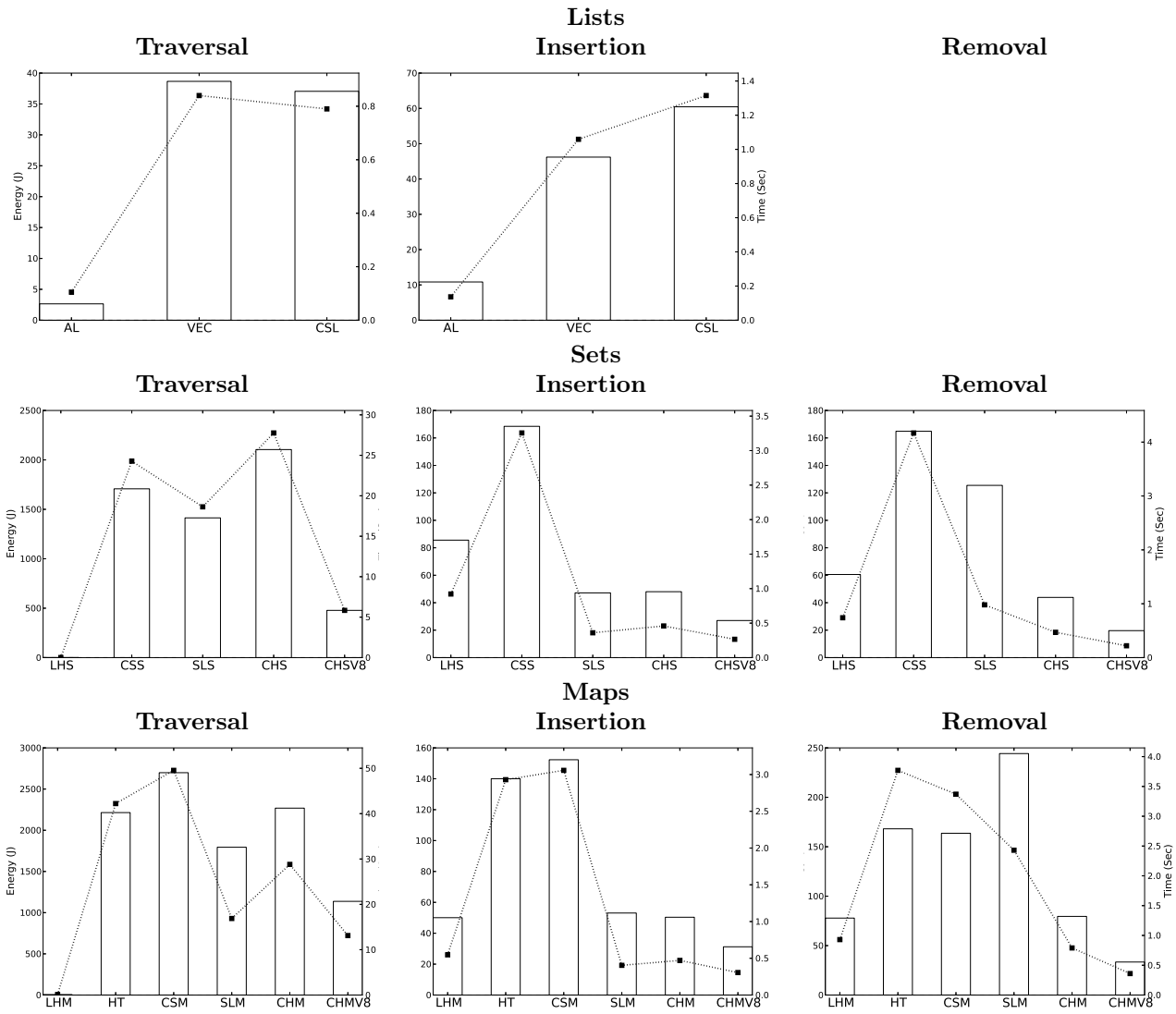

Figure 1: Energy and performance results for read, insertion and removal operations for different implementations of Java collections. Bars mean energy consumption and line means execution time. For the `List` figures, `AL` means `ArrayList`, `VEC` means `Vector`, and `CSL` means `Collections.synchronizedList()`. For the `Set` figures, `LSH` means `LinkedHashSet`, `CSS` means `Collections.synchronizedSet()`, `SLS` means `ConcurrentSkipListSet`, `CHS` means `ConcurrentHashSet`, and `CHSV8` means `ConcurrentHashSetV8`. Finally, for the `Map` figures, `LSM` means `LinkedHashMap`, `HT` means `Hashtable`, `CSM` means `Collections.synchronizedMap()`, `SLM` means `ConcurrentSkipListMap`, `CHM` means `ConcurrentHashMap`, and `CHMV8` means `ConcurrentHashMapV8`. We did not present the results for `CopyOnWriteArrayList` and `CopyOnWriteHashSet` in this figure because they present a much higher energy and time consumption, and they biased the understanding of the figures.

loop, for instance, using `for (String e: list)`, which is translated to an `Iterator` at compile time. In this configuration, `Vector` need to synchronize in two different moments: during the creation of the `Iterator` object, and in every call of the `next()` method. By contrast, the `Collections.synchronizedList()` does not synchronize on the `Iterator`, and thus has similar performance and energy usage when compared to our baseline, `ArrayList`; Energy decreased from 37.07J to 2.65J, whereas time decreased from 0.81 to 0.10. According to the `Collections.synchronizedList()` documentation, the programmer must ensure external synchronization when using `Iterator`.

Interestingly, however, we have observed that removals consumed 198.99x more energy than insertions on the `Vector` implementation. Time increased 853.21x more. This huge difference prevented us to conduct the experiments for all implementations in this configuration. We believe that this is because each call to the `List.remove()` method leads to a call of a `System.arrayCopy()` method in order to resize the `List`, since all these implementations of `List` are built upon arrays. In comparison, insertion operations only lead to a `System.arrayCopy()` call when the maximum number of elements is reached.

For all above cases, we observed that energy follows the same shape as time. At the first impression, this finding might seem to be "boring". However, recent studies have observed that energy and time are often not correlated [6, 11, 29], particularly true for concurrent applicaitonsFor this set of benchmarks, however, we believe that developers can safely use time as a proxy for energy, which can be a great help when refactoring an application to consume less energy.

Sets. First, for all of the implementations of `Set`, we can also observe that energy consumption follows the same behavior of execution time on traversal operations. For insertion and removal operations, they are not proportional. For all operations, the `ConcurrentHashMapV8` present the best results among the thread-safe ones. However, an interesting trade-off can be observed when performing traversal operations. As expected, the non-thread-safe implementation, `LinkedHashSet`, achieved the best energy consumption and execution time results, followed by the `CopyOnWriteArraySet` implementation. We believe that the same recommendation for `CopyOnWriteArrayList` fits here: this collection should only be used in scenarios where reads are the much more frequent than insertions. Interestingly, `ConcurrentHashMap` presented the worst results, consuming 1.23x more energy and 1.14x more time than.

Another interesting result is observed with `ConcurrentSkipListSet`, which consumes only 1.31x less energy than a `Collections.synchronizedList()` on removal operations, although it saves 4.25x in execution time. Internally, `ConcurrentSkipListSet` relies on a `ConcurrentSkipListMap`, which is non-blocking, linearizable, and based on the compare-and-swap (CAS) operation. During traversal, this collection marks the "next" pointer to keep track of triples (predecessor, node, successor) in order to detect when and how to unlink deleted nodes. Also, because of the asynchronous nature of these maps, determining the current number of elements (used in the `Iterator`) requires a traversal of all elements. These behaviors are susceptible to create the energy consumption overhead observed in Figure 1.

Maps. The `Map` implementations present a different picture. For the `LinkedHashMap`, `Hashtable` and `Collections.synchronizedMap()` implementations, energy follows the same curve as time, for both read and insertion operations, with the best results also achieved by the non-thread-safe implementation, `LinkedHashMap`. Surprisingly, however, the same cannot be said for the removal operations. Removal operations on `Hashtable` and `Collections.synchronizedMap()` exhibited energy consumption are proportionally higher than their execution time.

On the other hand, for the `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` implementations, more power is being consumed behind the scenes. Since energy consumption is the product of power consumption and time, if the benchmark receives a 1.5x speed-up but, at the same time, yields a threefold increase in power consumption energy consumption increase twofold. This scenario is roughly what happens in traversal operations, when transitioning from `Hashtable` to `ConcurrentHashMap`. Even though `ConcurrentHashMap` produces a speedup of 1.46x over the `Hashtable` implementation, it achieves that by consuming 1.50x more power. As a result, overall, `ConcurrentHashMap` consumed

slightly more energy than `Hashtable` (2.38%). This result is relevant mainly because several textbooks [30], research papers [31] and internet blog posts [32] suggest `ConcurrentHashMap` as the *de facto* replacement for the old associative `Hashtable` implementation. Our result suggests that the decision on whether or not to use `ConcurrentHashMap` should be made with care, in particular, in scenarios where the energy consumption is more important than performance. However, the newest `ConcurrentHashMapV8` implementation, released in the version 1.8 of the Java programming language, handles large maps or maps that have many keys with colliding hash codes more gracefully. `ConcurrentHashMapV8` provides a performance saving of 2.19x when compared to `ConcurrentHashMap`, and an energy saving of 1.99x in traversal operations (these savings are, respectively, 1.57x and 1.61x in insertion operations, and 2.19x and 2.38x in removal operations).

`ConcurrentHashMapV8` is a complete rewritten version of its predecessor. The primary design goal of this implementation is to maintain concurrent readability (typically method `get()`, but also on `Iterators`) while minimizing update contention. This map acts as a binned hash table. Internally, it uses tree-map-like structures to maintain bins containing more nodes than would be expected under ideal random key distributions over ideal numbers of bins. This tree also require an additional locking mechanism. While list traversal is always possible by readers even during updates, tree traversal is not, mainly because of tree-rotations that may change the root node and its links. Insertion of the first node in an empty bin is performed with a Compare-And-Set operation. Other update operations (insert, delete, and replace) require locks. Locking support for these locks relies on builtin "synchronized" monitors.

Energy-Performance Trade-offs. We used a well-known metric, *Energy* × *DelayProduct* (EDP) [33], in order to investigate the relationship between energy and performance. We compute the EDP for the benchmarks, with results presented in Table 1, where a smaller EDP value indicates the more favorable trade-off (*e.g.*, better energy efficiency). We use **boldface** to highlight the smallest value for each case.

From this table, we can observe that the non-thread-safe implementation is generally more favorable for energy-performance trade-offs than its thread-safe counterparts. This is particularly true for traversal operations, where the non-thread-safe implementations are the best for all groups of collections. For insertion and removal operations on `Lists`, the non-thread-safe implementation also achieves the best energy-performance. For the other cases, the best results are achieved by the `ConcurrentHashSetV8` and `ConcurrentHashMapV8` implementations, due to the considerable speedups achieved by these implementations in the presence of multiple threads.

Collections	EDP		
	TR	**IN**	**RM**
`AL`	**2.65**	**1.48**	—
`VEC`	38.66	48.92	—
`CSL`	37.05	79.49	—
`COW`	2.71	1,675,167.56	—
`LHS`	**0.48**	78.89	44.80
`CSS`	41,452.23	548.50	687.46
`SLS`	26,342.24	16.97	122.95
`COS`	3.04	3,124,257.43	—
`CHS`	58,435.88	22.04	20.61
`CHSV8`	2,792.22	**7.18**	**4.32**
`LHM`	**0.51**	27.38	72.38
`HT`	93,496.19	410.31	633.85
`CSM`	133,727.52	465.64	551.46
`SLM`	30,282.72	21.44	593.52
`CHM`	65,358.94	23.71	62.83
`CHMV8`	14,919.93	**9.55**	**12.02**

Table 1: EDP (a smaller value is better). We use the same abbreviations of Figure 1. TR means traversal, IN means insertion and RM means removal.

Maps tuning knobs. The `Map` implementations also have two important "tuning knobs": the *initial capacity* and *load factor*. The capacity is the total number of elements inside a `Map` and the initial capacity is the capacity at the time the `Map` is created. The default initial capacity size of most `Map` implementations is only 16 locations. We now report a set of experiments varying the initial capacity from 16 elements, 32, 320, 3200, 32000, 320000, 3200000, 3200000, and 32000000 — the last one is the total elements inside a collection. Figure 2 shows how energy consumption behaves using these different initial capacities configurations.

As we can observe from this figure, the results can vary greatly when using different inital capac-

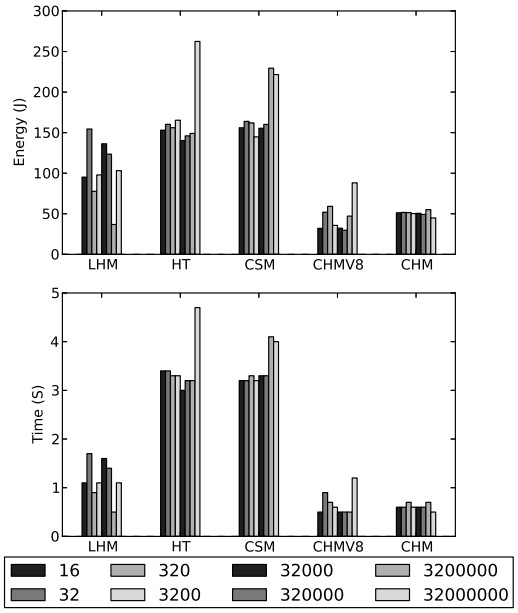

Figure 2: Energy consumption and performance variations with different initial capacities.

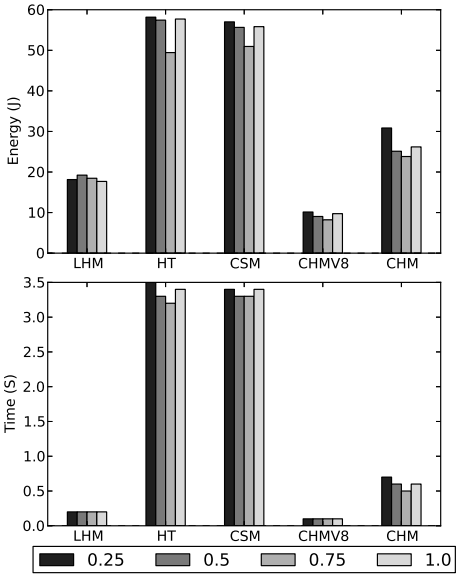

Figure 3: Energy consumption and performance variations with different load factors.

ities, both in energy consumption and execution time. The most evidente cases are when performing with a high number as a initial capacity in both `Hashtable` and `ConcurrentHashMap` implementations. `ConcurrentHashMapV8`, on the other hand, present a more stable results.

The other tuning knob is the load factor. It is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of elements inside a `Map` exceeds the product of the load factor and the current capacity, the hash table is rehashed; that is, its internal data structure is rebuilt. The default load factor value in most `Map` implementation is 0.75. It means that, using initial capacity as 16, and the load factor as 0.75, the product of capacity is 12 (16 * 0.75 = 12). Thus, after inserting the 12th key, the new map capacity after rehashing will be 32. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. Figure 3 shows how energy consumption behaves using different load factors configurations[5].

From this figure we can observe that, albeit small, the load factor also influences both energy consumption and time. For instance, when using a load factor of 0.25, we observed the most energy inefficient results, except in one case (the energy consumption of `LinkedHashMap`). We believe it is due to the successive times the map needs to be rehashed. Generally speaking, the default load factor (.75) offers a good tradeoff between performance, energy and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry, which can reflect in most of the `Map` operations, including `get()` and `put()`). It is possible to observe this cost when using a load factor of 1.0, which means that the map will be only rehashed when the number of current elements reaches the current maximum size. The maximum variation was found when performing a `Hastable`, in the default load factor, achieving 1.17x better energy consumption over the 0.25 configuration, and 1.09x in execution time.

Hash collisions. We also investigated how hash collisions impact energy consumption. In this scenario, a collision is a situation that occurs when two or more keys happen to have the same hashCode.

We performed experiments varying the number collisions from 20%, 50%, 70%, to 100% — when we have 100% of collisions, it means that all inserted keys used the same hashCode. However, we

[5]We did not performed experiments with `ConcurrentSkipListMap` because it does not provide access to initial capacity and load factor variables.

only experienced significant variation in energy consumption where the number of duplicated keys are more than 50%. When the percentage of colliding keys is above this threshold, both performance and energy consumption variations start being noticeable.

Figure 4 shows the extreme case, the 100% configuration. `LinkedHashMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` present an increment in performance and energy consumption (variations in energy and time, respectively: 10.90% and 33.33%, 78.84% and 96.15%, 12.32% and 68.75%), whereas `Hashtable` and `Collections.synchronizedMap()` present a decrement in both performance and energy (variations in energy and time, respectively: -10.97% and -18.51%, -25.12% and -32%).

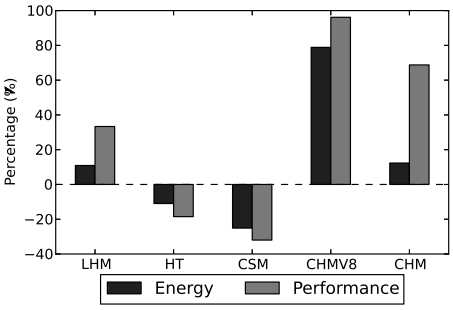

Figure 4: Energy consumption and performance variations when collision is 100%.

We believe this behavior can be explained in terms of how the `Map` deals with the duplicated keys. For instance, the `Hashtable` implementation does not take any care of additional keys, whereas `ConcurrentHashMap` and `ConcurrentHashMapV8` maintain a list of duplicated keys.

RQ1 Summary: *We observed that different implementations of the same collection can greatly impact both energy consumption and execution time. When comparing* `CopyOnWriteArrayList` *with the non-thread-safe implementation, the difference can be higher than 152x.*

RQ2 Summary: *We observed that different operations of the same collection can greatly impact both energy consumption and execution time. For instance, when performing with a* `Vector`*, a removal operation can consume about 200x more energy than a insertion one.*

5. Threat to Validity

We divide our discussion on threats to validity into internal factors and external factors.

Internal factors: First, the elements which we used are not randomly generated. We chose to not use random number generators because they can greatly impact the performance and energy consumption of our benchmarks. We observed standard deviation of over 70% between two executions when using the random number generators. We mitigate this problem by combining the index of the for loop plus the thread id that inserted the element. This approach also prevents compiler optimizations that may happen when using only the index of the for loop as the element to be inserted in the collection.

External factors: First, our results are limited by our selection of benchmarks. Nonetheless, our corpus spans a wide spectrum of collections, ranging from lists, sets, and maps. Second, there are other possible collections implementations beyond the scope of this paper. With our methodology, we expect similar analysis can be conducted by others. Third, our results are reported with the assumption that JIT is enabled. This stems from our observation that later runs of JIT-enabled executions do stabilize in terms of energy consumption and performance [6]. We experienced differences in standard deviation of over 30% when comparing the warmup run (first 3 executions) and later runs, but less than 5% when comparing the last 3 runs.

6. Conclusions

In this paper, we presented an empirical study that investigates the impacts of using different collections on energy usage. As subjects for the study, we analyzed the main methods of 16 types of commonly used collection in the Java language. The results of this study demonstrate that:

Based on these conclusions, there are several potential areas for future work. First, we plan on enlarging the scope of our study. Although we considered a significant number of subjects, adding additional collection, and their methods, would potentially allow us to refute or confirm some of our observations in addition to perform the removal experiments for all collections available. Second, we believe that other tuning knobs should be studied, such as varying the number of concurrent threads accessing the data structure, and varying the data

size being manipulated in the data structure. With insights of this study, we plan to introduce the concept of relaxed collection. One step towards this goal is to reduce their accuracy [34]. Since Java8 introduced the concept of `Streams`, which use implicitly parallelism and are well-suitable for data-parallel programs, an approximate solution for a given function, for instance `sum` the values of all elements, over a huge collection can take a fraction of memory, time and, last but not least, energy consumption.

7. Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. Gustavo is supported by CAPES/Brazil, and Fernando is supported by CNPq/Brazil (306619/2011- 3, 487549/2012-0 and 477139/2013-2), FACEPE/Brazil (APQ-1367-1.03/12) and INES (CNPq 573964/ 2008-4 and FACEPE APQ-1037-1.03/08). Any opinions expressed here are from the authors and do not necessarily reflect the views of the sponsors.

References

[1] A. Chandrakasan, S. Sheng, R. Brodersen, Low-power cmos digital design, Solid-State Circuits, IEEE Journal of 27 (4) (1992) 473–484.

[2] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, C. Le, Rapl: Memory power estimation and capping, in: ISLPED, 2010.

[3] H. Ribic, Y. Liu, Energy-efficient work-stealing language runtimes, in: ASPLOS, 2014.

[4] T. Bartenstein, Y. Liu, Rate types for stream programs, in: OOPSLA, 2014.

[5] D. Li, W. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in: GREENS, 2014.

[6] G. Pinto, F. Castor, Y. Liu, Understanding energy behaviors of thread management constructs, in: OOPSLA, 2014.

[7] Y.-W. Kwon, E. Tilevich, Reducing the energy consumption of mobile applications behind the scenes, in: ICSM, 2013, pp. 170–179.

[8] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: ESEM, 2014.

[9] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: MSR, 2014.

[10] C. Wilke, S. Richly, S. Gotz, C. Piechnick, U. Assmann, Energy consumption and efficiency in mobile applications: A user feedback study, in: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013, pp. 134–141.

[11] D. Li, S. Hao, W. G. J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: ISSTA, 2013.

[12] C. Seo, S. Malek, N. Medvidovic, Component-level energy consumption estimation for distributed java-based software systems, in: M. Chaudron, C. Szyperski, R. Reussner (Eds.), Component-Based Software Engineering, Vol. 5282 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 97–113.

[13] N. Hunt, P. S. Sandhu, L. Ceze, Characterizing the performance and energy efficiency of lock-free data structures, in: Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures, 2011.

[14] J. Li, J. F. Martínez, Power-performance considerations of parallel computing on chip multiprocessors, ACM Trans. Archit. Code Optim. 2 (4) (2005) 397–422.

[15] C. Bunse, H. Hpfner, S. Roychoudhury, E. Mansour, Energy efficient data sorting using standard sorting algorithms, in: J. Cordeiro, A. Ranchordas, B. Shishkov (Eds.), Software and Data Technologies, Vol. 50 of Communications in Computer and Information Science, Springer Berlin Heidelberg, 2011, pp. 247–260.

[16] C. Sahin, F. Cayci, I. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: GREENS, 2012, pp. 55–61.

[17] Y.-W. Kwon, E. Tilevich, Cloud refactoring: automated transitioning to cloud-based services, Autom. Softw. Eng. 21 (3) (2014) 345–372.

[18] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, S. Yang, Refactoring android java code for on-demand computation offloading, in: OOPSLA, 2012.

[19] T. Cao, S. M. Blackburn, T. Gao, K. S. McKinley, The yin and yang of power and performance for asymmetric hardware and managed software, in: ISCA, 2012.

[20] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, J. Clause., How does code obfuscations impact energy usage?, in: ICSME, 2014.

[21] N. Vallina-Rodriguez, J. Crowcroft, Energy management techniques in modern mobile handsets, Communications Surveys Tutorials, IEEE 15 (1) (2013) 179–198.

[22] M. Fomitchev, E. Ruppert, Lock-free linked lists and skip lists, in: PODC, 2004.

[23] B.-U. Pagel, H.-W. Six, H. Toben, P. Widmayer, Towards an analysis of range query performance in spatial data structures, in: PODS, 1993.

[24] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: PODC, 2003.

[25] W. Torres, G. Pinto, B. Fernandes, J. Oliveira, F. Ximenes, F. Castor, Are java programmers transitioning to multicore?: A large scale study of java floss, in: TMC, 2011.

[26] Y. Lin, D. Dig, Check-then-act misuse of java concurrent collections, in: ICST, 2013.

[27] G. Xu, Coco: Sound and adaptive replacement of java collections, in: ECOOP, 2013.

[28] I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: ICSE, 2014.

[29] A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444 – 449, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011. doi:http://dx.doi.org/10.1016/j.jocs.2013.01.005.
URL http://www.sciencedirect.com/science/article/pii/S1877750313000173

[30] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, Java Concurrency in Practice, Addison-Wesley Professional, 2005.

[31] D. Dig, J. Marrero, M. D. Ernst, Refactoring sequential java code for concurrency via concurrent libraries, in: ICSE, 2009.

[32] B. Goetz, Java theory and practice: Concurrent collections classes, http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html, accessed: 2014-09-29.

[33] J. Laros III, K. Pedretti, S. Kelly, W. Shu, K. Ferreira, J. Vandyke, C. Vaughan, Energy delay product, in: Energy-Efficient High Performance Computing, SpringerBriefs in Computer Science, Springer London, 2013, pp. 51–55.

[34] M. Carbin, D. Kim, S. Misailovic, M. Rinard, Proving acceptability properties of relaxed nondeterministic approximate programs, in: PLDI, 2012.